Dynodroid: An Input Generation System for Android Apps

Aravind MacHiry Rohan Tahiliani Mayur Naik

Georgia Institute of Technology
{amachiry, rohan_tahil, naik}@gatech.edu

ABSTRACT

We present a system Dynodroid for generating relevant inputs to unmodified Android apps. Dynodroid views an app as an event-driven program that interacts with its environment by means of a sequence of events through the Android framework. By instrumenting the framework once and for all, Dynodroid monitors the reaction of an app upon each event in a lightweight manner, using it to guide the generation of the next event to the app. Dynodroid also allows interleaving events from machines, which are better at generating a large number of simple inputs, with events from humans, who are better at providing intelligent inputs.

We evaluated Dynodroid on 50 open-source Android apps, and compared it with two prevalent approaches: users manually exercising apps, and Monkey, a popular fuzzing tool. Dynodroid, humans, and Monkey covered 55%, 60%, and 53%, respectively, of each app's Java source code on average. Monkey took 20X more events on average than Dynodroid. Dynodroid also found 9 bugs in 7 of the 50 apps, and 6 bugs in 5 of the top 1,000 free apps on Google Play.

1. INTRODUCTION

Mobile apps—programs that run on advanced mobile devices such as smartphones and tablets—are becoming increasingly prevalent. Unlike traditional enterprise software, mobile apps serve a wide range of users in heterogeneous and demanding conditions. As a result, mobile app developers, testers, marketplace auditors, and ultimately end users can benefit greatly from what-if analyses of mobile apps.

What-if analyses of programs are broadly classified into static and dynamic. Static analyses are hindered by features commonly used by mobile apps such as code obfuscation, native libraries, and a complex SDK framework. As a result, there is growing interest in dynamic analyses of mobile apps (e.g., [2,13,14,26]). A key challenge to applying dynamic analysis ahead-of-time, however, is obtaining program inputs that adequately exercise the program's functionality.

We set out to build a system for generating inputs to mobile apps on Android, the dominant mobile app platform, and identified five key criteria that we felt such a system must satisfy in order to be useful:

- *Robust*: Does the system handle real-world apps?
- *Black-box*: Does the system forgo the need for app sources and the ability to decompile app binaries?
- *Versatile*: Is the system capable of exercising important app functionality?
- *Automated*: Does the system reduce manual effort?
- *Efficient*: Does the system generate concise inputs, i.e., avoid generating redundant inputs?

This paper presents a system Dynodroid that satisfies the above criteria. Dynodroid views a mobile app as an event-driven program that interacts with its environment by means of a sequence of events. The main principle underlying Dynodroid is an *observe-select-execute* cycle, in which it first *observes* which events are relevant to the app in the current state, then *selects* one of those events, and finally *executes* the selected event to yield a new state in which it repeats this process. This cycle is relatively straightforward for *UI events*—inputs delivered via the program's user interface (UI) such as a tap or a gesture on the device's touchscreen. In the *observer* stage, Dynodroid determines the layout of widgets on the current screen and what kind of input each widget expects. In the *selector* stage, Dynodroid uses a novel randomized algorithm to select a widget in a manner that penalizes frequently selected widgets without starving any widget indefinitely. Finally, in the *executor* stage, Dynodroid exercises the selected widget.

In practice, human intelligence may be needed for exercising certain app functionality, in terms of generating both individual events (e.g., inputs to text boxes that expect valid passwords) and sequences of events (e.g., a strategy for winning a game). For this reason, Dynodroid allows a user to observe an app reacting to events as it generates them, and lets the user pause the system's event generation, manually generate arbitrary events, and resume the system's event generation. Our overall system thereby combines the benefits of both the automated and manual approaches.

We discussed how Dynodroid handles UI events, but significant functionality of mobile apps is controlled by non-UI events we call *system events*, such as an incoming SMS message, a request by another app for the device's audio, or a notification of the device's battery power running low. Satisfying our five desirable criteria in the presence of system events is challenging for two reasons. First, the number of possible system events is very large, and it is impractical to generate all possible permutations of those events. For instance, the Gingerbread version of Android supports 108 different broadcast receivers, each of which can send notifications called *intents* to an app. Second, many system events have structured data that must be constructed and dispatched correctly to the app alongside an intent. For example, generating an incoming SMS message event entails constructing and sending a suitable object of class `android.telephony.SmsMessage`.

A distinctive aspect of mobile apps is that all such apps, regardless of how diverse their functionality, are written against a common framework that implements a significant portion of the app's functionality. Dynodroid exploits this aspect pervasively in observing, selecting, and executing system events, as we describe next.

Dynodroid addresses the problem of handling a very large number of possible system events by using the observation that in practice, a mobile app typically reacts to only a small fraction of them we call relevant events. An event is *relevant* to an app if the app has registered a listener for the event with the framework. Determining when an app registers (or unregisters) a listener for a system event does not require modifying the app: it suffices to instrument the framework once and for all. The *observer* monitors the app's interaction with the framework via this instrumentation, and presents the *selector* only events that the app has registered to listen. Finally, the *executor* constructs any data associated with the selected event and dispatches it to the app. An important aspect of Dynodroid is that it constructs this data organically. For instance, the listener for network connectivity change events expects an object of class `android.net.NetworkInfo` describing the new status of the network interface. Simulating this event in an Android emulator is delicate as Dynodroid cannot arbitrarily create such objects; it must instead obtain them from a pool maintained by system service `android.net.ConnectivityManager`. Finally, whenever an event is executed, any previously relevant event may become irrelevant and vice versa, for the next observe-select-execute cycle.

We implemented Dynodroid for the Gingerbread version of Android and applied it to 50 diverse, real-world, open-source apps. We compared the performance of Dynodroid in terms of each app's Java source code coverage to two prevalent approaches: one involving expert Android users manually exercising these apps, and another using Monkey [8], a popular fuzzing tool for Android apps. Dynodroid, humans, and Monkey covered 55%, 60%, and 53% code, respectively, on average per app. Dynodroid was able to cover 83% of the code covered by humans per app on average, demonstrating its ability to automate testing. Also, Dynodroid achieved peak coverage faster than Monkey, with Monkey requiring 20X more events on average, showing the effectiveness of our randomized event selection algorithm. Finally, Dynodroid revealed 9 bugs in 7 of the 50 apps. In a separate experiment, we applied Dynodroid to the top 1,000 free apps in Google Play, and it exposed 6 bugs in 5 of those apps, demonstrating its robustness.

We summarize the main contributions of our work:

1. We propose an effective system for generating inputs to mobile apps. The system is based on a novel "observe-select-execute" principle that efficiently generates a sequence of relevant events. To adequately exercise app functionality, it generates both UI events and system events, and seamlessly combines events from human and machine.

2. We show how to observe, select, and execute system events for Android in a mobile device emulator without modifying the app. The central insight is to tailor these tasks to the vast common framework against which all apps are written and from which they primarily derive their functionality.

3. We present extensive empirical evaluation of the system, comparing it to the prevalent approaches of manual testing and automated fuzzing, using metrics including code coverage and number of events, for diverse Android apps including both open-source apps and top free marketplace apps.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the overall architecture of our system. The next three sections present its three main parts: Section 4 the executor, Section 5 the observer, and Section 6 the selector. Section 7 presents our experimental results. Section 8 discusses limitations and finally Section 9 concludes.

2. RELATED WORK

There are broadly three kinds of approaches for generating inputs to mobile apps: *fuzz testing*, which generates random inputs to the app; *systematic testing*, which systematically tests the app by executing it symbolically; and *model-based testing*, which tests a model of the app. We elaborate upon each of these three approaches in this section.

Fuzz Testing. The Android platform includes a fuzz testing tool Monkey [8] that generates a sequence of random UI events to unmodified Android apps in

a mobile device emulator. Recent work has applied Monkey to find GUI bugs [16] and security bugs [19] in apps. Fuzz testing is a black-box approach, it is easy to implement robustly, it is fully automatic, and it can efficiently generate a large number of simple inputs. But it is not suited for generating inputs that require human intelligence (e.g., constructing valid passwords, playing and winning a game, etc.) nor is it suited for generating highly specific inputs that control the app's functionality, and it may generate highly redundant inputs. Finally, Monkey only generates UI events, not system events. It would be challenging to randomly generate system events given the large space of possible such events and highly structured data that is associated with many of them.

Systematic Testing. Several recent efforts [10, 17, 23] have applied symbolic execution [12, 15, 18] to generate inputs to Android apps. Symbolic execution automatically partitions the domain of inputs such that each partition corresponds to a unique program behavior (e.g., execution of a unique program path). Thus, it avoids generating redundant inputs and can generate highly specific inputs, but it is difficult to scale due to the notorious path explosion problem. Moreover, symbolic execution is not black-box and requires heavily instrumenting the app in addition to the framework.

Model-based Testing. Model-based testing has been widely studied in testing GUI-based programs [11, 21, 22, 27] using the GUITAR framework [5]. GUITAR has been applied to Android apps, IPhone apps, and web apps, among others. In general, model-based testing requires users to provide a model of the app's GUI [25, 28], though automated GUI model inference tools tailored to specific GUI frameworks exist as well [9, 24]. One such tool in the Android platform, that Dynodroid also uses for observing relevant UI events, is Hierarchy Viewer [6]. Model-based testing harnesses human and framework knowledge to abstract the input space of a program's GUI, and thus reduce redundancy and improve efficiency, but existing approaches predominantly target UI events as opposed to system events.

3. SYSTEM ARCHITECTURE

This section presents the system architecture of Dynodroid. Algorithm 1 shows the overall algorithm of Dynodroid. It takes as input the number n of events to generate to an app under test. It produces as output a list L of n events it generates. The first generated event is to install and start the app in a mobile device emulator. The remaining $n-1$ events are generated one at a time in an observe-select-execute cycle. Each of these events constitutes either a UI input or non-UI input to the app, which we call *UI event* and *system event*, respectively. The two kinds of events are conceptually alike: each event of either kind has a type and associated data. We distinguish between them because, as we explain below, Dynodroid uses different mechanisms to handle them. The EXECUTOR executes the current event, denoted e, in the current emulator state, denoted s, to yield a new emulator state that overwrites the current state. Next, the OBSERVER computes which events are relevant in the new state. We denote the set of relevant events E. Finally, the SELECTOR selects one of the events from E to execute next, and the process is repeated.

Algorithm 1 Overall algorithm of Dynodroid.

```
INPUT: Number n > 0 of events to generate.
OUTPUT: List L of n events.
L := empty list
e := event to install and start app under test
s := initial program state
for i from 1 to n do
  append e to L
  // Execute event e in current state s to yield updated state.
  s := EXECUTOR(e, s)
  // Compute set E of all events relevant in current state s.
  E := OBSERVER(s)
  // Select an event e ∈ E to execute in next iteration.
  e := SELECTOR(E)
end for
```

Figure 1 presents a dataflow diagram of Dynodroid that provides more details about the mechanisms it uses to implement the EXECUTOR, the OBSERVER, and the SELECTOR on the Android platform.

The EXECUTOR triggers a given event using the appropriate mechanism based on the event kind. It uses the Android Debug Bridge (`adb`) to send the event to an Android device emulator that is running the app under test. For UI events, the ADB host talks to the ADB daemon (`adbd`) on the emulator via the `monkeyrunner` tool. Note that this tool, which is used to send events to an emulator via an API, is unrelated to the Monkey fuzz testing tool, which runs in an `adb` shell directly on the emulator. For system events, the ADB host talks to the Activity Manager tool (`am`) on the emulator, which can send system events as *intents* to running apps. Section 4 describes the EXECUTOR in further detail.

The OBSERVER computes the set of events that are relevant to the app under test in the current emulator state. It consists of two parts to handle the two kinds of events: the Hierarchy Viewer tool for UI events, and the instrumented framework (SDK) for system events. Hierarchy Viewer provides information about the app's GUI elements currently displayed on the device's screen. The OBSERVER computes the set of relevant UI events E_1 from this information. The instrumented SDK provides information about broadcast receivers and system services for which the app is currently registered. The OBSERVER computes the set of relevant system events

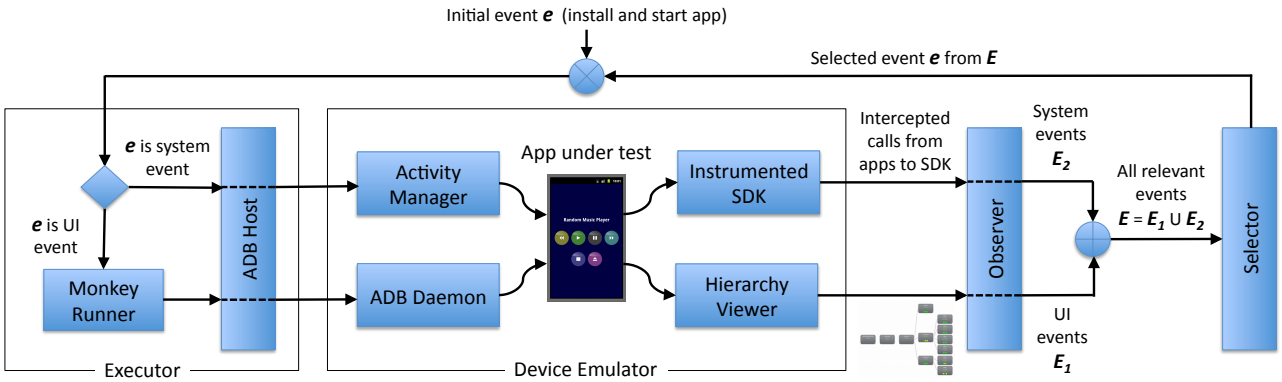


Figure 1: Dataflow diagram of Dynodroid for the Android platform.

E_2 from this information. It provides the set of all relevant events $E = E_1 \cup E_2$ to the SELECTOR. Section 5 provides more details about the OBSERVER.

The SELECTOR selects an event from E as the next event to be triggered. Dynodroid implements various selection strategies that one can choose from upfront, including deterministic vs. randomized, and history dependent vs. history oblivious strategies. Section 6 describes these strategies and Section 7 evaluates them on a code coverage client for 50 open-source apps.

4. EXECUTOR

Dynodroid allows both machine and human to generate events in order to combine the benefits of automated and manual input generation. Figure 2 shows how Dynodroid allows switching between machine and human. The EXECUTOR listens to commands from a console and starts in *human mode*, in which it does not trigger any events and instead allows the human to exercise the app uninterrupted in the emulator, until a RESUME command is received from the console. At this point, the EXECUTOR switches to *machine mode* and generates events until the given bound n is reached or a PAUSE command is received from the console. In the latter case, the EXECUTOR switches to *human mode* again. Human inputs do not count towards the bound n. Also, nothing prevents the human from exercising the app in *machine mode* as well, along with the machine. Finally, the STOP command from the console stops Dynodroid.

The EXECUTOR executes an event, chosen by the SELECTOR, on the emulator via the Android Debug Bridge (`adb`). It uses separate mechanisms for executing UI events and system events, as described next.

User events are triggered using the `monkeyrunner` tool. This tool is a wrapper around `adb` that provides an API to execute UI events. We wrote a python script that takes an input command over a socket connection, executes the command on the emulator using `monkeyrunner`, and returns the result of the execution back over the socket. The script can send UI events such

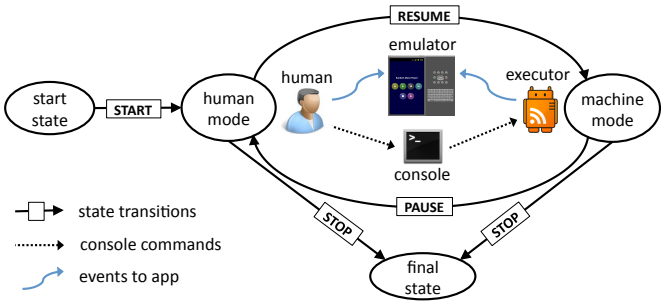


Figure 2: State transition diagram of Dynodroid.

as taps, gestures, and text inputs (see Section 5.1).

System events are triggered using the Activity Manager tool (`am`). Broadcast events of any type can be triggered using this tool provided the correct arguments are supplied. The `am` tool is incapable of handling arguments containing custom data in binary format. We modified the tool to handle binary data so that broadcast events can be triggered with the required data. We add extra information before triggering specific broadcast events, e.g., the phone number and message text data for an SMS_RECEIVED event (see Section 5.2).

5. OBSERVER

The OBSERVER computes the set of relevant events after an event is executed. We consider an event *relevant* if triggering that event may result in executing code that is part of the app. The goal of the OBSERVER is to efficiently compute as small a set of relevant events as possible without missing any. This section describes how the OBSERVER computes relevant UI events (Section 5.1) and relevant system events (Section 5.2).

5.1 UI Events

These are events generated by the SDK in response to interaction by users with the device's input mechanisms. Dynodroid supports two input mechanisms:

	Relevant Event			
	Tap	**LongTap**	**Drag**	**Text**
If app registers callback:				
onClickListener	✓			
onLongClickListener		✓		
onTouchListener	✓	✓	✓	
onKeyListener				✓
onCreateContextMenuListener		✓		
If app overrides method:				
onTouchEvent	✓	✓	✓	
performLongClick		✓		
performClick	✓			
onKeyDown				✓
onKeyUp				✓

Table 1: Mapping registered callback methods and overridden methods to relevant UI events.

touchscreen and navigation buttons (specifically, "back" and "menu" buttons). We found these sufficient in practice, for three reasons: they are the most common input mechanisms, the mechanisms we do not support (keyboard, trackball, etc.) are device-dependent, and there is often redundancy among different input mechanisms.

The OBSERVER analyzes the app's current UI state in order to compute relevant UI events. First, it deems clicking each navigation button as a relevant UI event, since these buttons are always enabled. Second, it inspects the *view hierarchy*, which is an Android-specific tree representation of the app's UI currently displayed on the touchscreen. Each node of the tree is an object of (some subclass of) View, the base class of all UI elements (buttons, text boxes, etc.). Each non-leaf node is an object of ViewGroup and serves as an invisible layout containing other Views (or other ViewGroups) that are represented as its children in the view hierarchy. The SDK provides two ways by which an app can react to inputs to a UI element: by overriding a method of the corresponding View object's class, or by registering a callback with it.

The SDK dispatches each input on the touchscreen to the root node of the view hierarchy, which in turn depending on the position of the input dispatches it recursively to one of its children, until the view to which the input was intended executes a callback and returns true, denoting that the input was successfully handled. The OBSERVER obtains the view hierarchy from a service called ViewServer that runs on Android devices having debug support. Once it obtains the view hierarchy, it considers only the View objects at leaf nodes of the tree as interesting, as these correspond to visible UI elements that users can interact with. It extracts two kinds of data from each such object about the corresponding UI element: (a) the set of callback methods registered and the set of methods overridden by the app, for listening to inputs to this UI element, and (b) the location and size of the UI element on the touchscreen (the position of its top left corner, its width, height, and scaling factor). The native Hierarchy Viewer does not provide all of the above data; we modified the Android SDK source to obtain it. The OBSERVER uses the data in item (a) to compute which UI events are relevant, as dictated by Table 1. It supports all common touchscreen inputs: tap inputs, limited kinds of gestures, and text inputs. Finally, the OBSERVER uses the data in item (b) to compute the parameters of each such event, as dictated by Table 2.

5.2 System Events

These are events generated by the SDK in response to non-UI inputs, such as an incoming phone call, a change in geo-location, etc. The SDK provides one of two means by which an app can receive each type of system event: *broadcast receiver* and *system service*. In either case, the app registers a callback to be called by the SDK when the event occurs. The app may later unregister it to stop receiving the event.

The OBSERVER uses the same mechanism for extracting relevant system events via broadcast receivers and system services: it instruments the SDK to observe when an app registers (or unregisters) for each type of system event. This instrumented SDK is a file system.img that is loaded on the emulator during bootup. It is produced once and for all by compiling Java source code of the original SDK that is manually modified to inject the instrumentation. A system event becomes relevant when an app registers to receive it and, conversely, it becomes irrelevant when an app unregisters it. As in the case of UI events, the OBSERVER computes not only which system events are relevant, but also what data to associate with each. Unlike for UI events, however, this data can be highly-structured SDK objects (instead of primitive-typed data). We next outline how the OBSERVER handles specific broadcast receivers (Section 5.2.1) and system services (Section 5.2.2).

5.2.1 Broadcast Receiver Events

Android provides two ways for an app to register for system events via a broadcast receiver, depending on the desired lifetime: dynamically or statically. In the dynamic case, the receiver's lifetime is from when Context.registerReceiver() is called to either until Context.unregisterReceiver() is called or until the lifetime of the registering app component. In the static case, the receiver is specified in file AndroidManifest.xml, and has the same lifetime as the app. In either case, the

Event Type	Parameters	Description
Tap	Tap($l + w/2$, $t + h/2$)	trigger Tap at center of view
LongTap	LongTap($l + w/2$, $t + h/2$)	trigger LongTap at center of view
Drag	random one of: Drag(l, t, l+w, t+h), Drag(l+w, t+h, l, t), Drag(l, t+h, l+w, t), Drag(l+w, t, l, t+h), Drag(l, t+$h/2$, l+w, t+$h/2$), Drag(l+w, t+$h/2$, l, t+$h/2$), Drag(l+$w/2$, t, l+$w/2$, t+h), Drag(l+$w/2$, t+h, l+$w/2$, t)	randomly trigger one of gestures: TL to BR, BR to TL, BL to TR, TR to BL, ML to MR, MR to ML, MT to MB, MB to MT
Text	arbitrary fixed string	trigger arbitrary text input

Table 2: User event parameters: l, t, w, h denote left position, top position, width, and height of the view, scaled by the view's scaling factor. TL, BR, BL, TR, MB, MT, ML, MR denote top left, bottom right, bottom left, top right, mid bottom, mid top, mid left, and mid right points of the view.

receiver defines a callback method overriding BroadcastReceiver.onReceive() that the SDK calls when the event occurs (we provide an example below). The OBSERVER supports both kinds of receivers.

The Gingerbread SDK version we instrumented has 108 different kinds of system events as *intents* for which an app may register via a broadcast receiver. For proof of concept, we chose 25 of them as follows: we counted the number of top 1,000 free apps in the Google Play market that statically register a broadcast receiver for each intent, and we chose those intents that was registered in this manner by 10 or more apps. Supporting additional intents is straightforward: it involves identifying the type of data associated with the intent and providing valid values for the data. The data includes an optional URI, and a Bundle object which is a key-value map that contains any extra information. Finally, for statically registered receivers, we also explicitly identify the receiver in the intent, since unlike UI events which are dispatched to a single View object, broadcast intents are by default dispatched to all receivers (possibly from several apps) that register for them.

Table 3 shows the type and values of the data used by the OBSERVER for our 25 chosen broadcast intents. For brevity we abbreviate the name of each intent's action, e.g., using SMS_RECEIVED instead of "android.provider.Telephony.SMS_RECEIVED". All the data shown, except those of type URI, are provided in a Bundle object. For instance, an app may register the following receiver for the SMS_RECEIVED intent which is broadcast upon incoming SMS messages:

```
public class SmsReceiver extends BroadcastReceiver {
  @Override public void onReceive(Context c, Intent e) {
    Bundle b = e.getExtras();
    Object[] p = (Object[]) b.get("pdus");
    SmsMessage[] a = new SmsMessage[p.length];
    for (int i = 0; i < pdus.length; i++)
       a[i] = SmsMessage.createFromPdu((byte[]) p[i]);
  }
};
```

To trigger this event, the EXECUTOR serializes the appropriate intent along with a Bundle object that has a key named "pdus" mapped to a byte array denoting an array of SmsMessage objects. We supply an array of a single SmsMessage object with an arbitrary but well-formed phone number and message text.

5.2.2 System Service Events

System services are a fixed set of processes that provide abstractions of different functionality of an Android device. We distinguish services whose offered functionality depends on app-provided data from those that are independent of such data. We call such services *internally* vs. *externally* triggered, as they depend on data internal or external to the app under test. For instance, the AlarmManager service is internally triggered, as it depends on the alarm duration given by an app, but the LocationManager service is externally triggered, as it depends on the device's geo-location. Dynodroid only controls externally triggered services as the app itself controls internally triggered services.

Table 4 shows how Dynodroid handles events of each externally triggered service. The "Register/Unregister Mechanism" shows how an app registers or unregisters for the service, which the OBSERVER observes via SDK instrumentation. Since services are global components, the OBSERVER uses the ID of the app under test to filter out observing other apps that may also register or unregister for these services. The "Callback Mechanism" shows how the app specifies the callback to handle events by the service. Lastly, the "Trigger Mechanism" shows how the EXECUTOR triggers the callback, usually via a command sent from the ActivityManager tool running on the emulator (see Section 4).

The following example showing how an app may use the LocationManager service:

```
GpsStatus.Listener l = new GpsStatus.Listener() {
  @Override public void onGpsStatusChanged(int event) {...}
};
LocationManager lm = getSystemService(LOCATION_SERVICE);
lm.addGpsStatusListener(l);
...
lm.removeGpsStatusListener(l);
```

From the point at which the app registers to listen to GPS status changes by calling addGpsStatusListener()

Action Name	Data Type : Description	Data Value
APPWIDGET_UPDATE	int[] : IDs of App Widgets to update	random (1-10) sized array of random (0-1000) ints
CONNECTIVITY_CHANGE	android.net.NetworkInfo : status of the network interface	random NetworkInfo object from android.net.ConnectivityManager
PACKAGE_ADDED	int : uid assigned to new package, bool : true if this follows a 'removed' broadcast for the same package	uid of random installed package, random bool
PACKAGE_REMOVED	int : uid previously given to package, bool : true if removing entire app, bool : true if an 'added' broadcast for the same package will follow	uid of random installed package, random bool, random bool
PACKAGE_REPLACED	int : uid assigned to new package	uid of random installed package
MEDIA_MOUNTED	URI : path to mount point of media, bool : true if media is read-only	"/mnt/sdcard", false
TIMEZONE_CHANGED	TimeZone : time-zone representation	America/Los_Angeles time-zone
MEDIA_BUTTON	android.view.KeyEvent : key event that caused this broadcast	KeyEvent object with action as ACTION_UP and value as KEYCODE_MEDIA_PLAY_PAUSE
SMS_RECEIVED	android.telephony.SmsMessage[] : array of received SMS messages	array of 1 SmsMessage object with arbitrary MSISDN and message
MEDIA_UNMOUNTED	URI : path to mount point of media	"/mnt/sdcard"
PHONE_STATE	int : phone state (idle|ringing|offhook), String : incoming phone number	1 (ringing), an arbitrary MSISDN
MEDIA_SCANNER_FINISHED	URI	"/mnt/sdcard"
NEW_OUTGOING_CALL	String: outgoing phone number	an arbitrary MSISDN
[BATTERY_[CHANGED|LOW|OKAY] | ACTION_POWER_[DIS]CONNECTED | ACTION_SHUTDOWN | TIME_SET | AUDIO_BECOMING_NOISY | DATE_CHANGED | USER_PRESENT | MEDIA_EJECT | BOOT_COMPLETED]	none	none

Table 3: Broadcast receiver events with associated data as implemented in Dynodroid.

to the point at which it unregisters by calling removeGpsStatusListener(), the OBSERVER regards GPS status change as a relevant system event. If the SELECTOR described in the next section selects this event, then the EXECUTOR triggers it by sending telnet command "geo fix G" to the emulator, where G is an arbitrary geo-location (a triple comprising a latitude, longitude, and altitude). This in turn results in invoking callback onGpsStatusChanged() defined by the app.

6. SELECTOR

The SELECTOR selects an event for the EXECUTOR to execute from the set of relevant events E computed by the OBSERVER. We implemented three different selection strategies in the SELECTOR, called Frequency, UniformRandom, and BiasedRandom. This section describes these strategies.

The Frequency strategy selects an event from E that has been selected least frequently by it so far. The rationale is that infrequently selected events have a higher chance of exercising new app functionality. A drawback of this strategy is that its deterministic nature leads the app to the same state in repeated runs. In practice, different states might be reached in different runs because of non-determinism inherent in dynamic analysis of Android apps, due to factors such as concurrency and asynchrony (see Section 8); however, we cannot rely on them to cover much new app functionality.

The UniformRandom strategy circumvents the above problem by selecting an event from E uniformly at random. This is essentially the strategy used by the Monkey fuzz testing tool, with three key differences. First, Monkey can only generate UI events, preventing it from covering app functionality controlled by system events. Second, Monkey does not compute relevant events and can send many events that are no-ops in the current state, hindering efficiency and conciseness of the generated event sequence. Third, Monkey does not compute

Service	Register/Unregister Mechanism	Callback Mechanism	Trigger Mechanism
Audio-Manager	registerMediaButton-EventReceiver(C) / unregisterMediaButton-EventReceiver()	invoke component denoted by ComponentName object C	send KeyPress event with keycode of randomly chosen MEDIA_BUTTON via `monkeyrunner`
	requestAudioFocus(L) / abandonAudioFocus()	call onAudioFocusChange() in AudioManager.OnAudioFocus-ChangeListener object L	simulate incoming phone call from a fixed MSIDN N via telnet command "gsm call N"
Location-Manager	addGpsStatusListener(L) / removeGpsStatusListener()	call onGpsStatusChanged() in GpsStatus.Listener object L	set geo-location to fixed value G via telnet command "geo fix G"
	addNmeaListener(L) / removeNmeaListener()	call onNmeaReceived() in GpsStatus.NmeaListener object L	send fixed NMEA data S via telnet command "geo nmea S"
	addProximityAlert(G, P) / removeProximityAlert()	trigger PendingIntent P	set geo-location to registered proximal value G via telnet command "geo fix G"
	requestLocationUpdates() / removeUpdates()	if LocationListener specified, call on[Location|Status]Changed() or onProvider[Enabled|Disabled](); else call PendingIntent or post to message queue of given Looper	set geo-location twice, via commands "geo fix G_1" and "geo fix G_2"; G_1 is random geo-location, G_2 is based on G_1 and registered criteria.
	requestSingleUpdate() / auto unregister after update	same as above	same as above
Sensor-Manager	registerListener(L, S) / unregisterListener()	call on[Accuracy|Sensor]Changed() on SensorEventListener object L	set random values x, y, z for sensor S via telnet command "sensor set S x:y:z"
Telephone-Manager	listen(L, S), with state S non-zero / zero	call onDataActivity() or any of several on*Changed() methods on PhoneStateListener object L	trigger state change via telnet command "gsm C D"; C is random gsm command and D random valid data.

Table 4: Handling of events of externally triggerable system services in Dynodroid.

a model of the app's UI, which has pros and cons. On one hand, it prevents Monkey from identifying *observationally equivalent* UI events (e.g., taps at different points of the same button that have the same effect, of clicking the button) and hinders efficiency and conciseness; on the other hand, Dynodroid sends mostly fixed inputs (see Table 2) to a widget, and may fail to adequately exercise custom widgets (e.g., a game that interprets taps at different points of a widget differently).

A drawback of the UniformRandom strategy is that it does not take any domain knowledge into account: it does not distinguish between UI events and system events, nor between different contexts in which an event may occur, nor between frequent and infrequent events. For instance, an event that is always relevant (e.g., an incoming phone call event) stands to be picked disproportionately more often than an event that is relevant only in certain contexts (e.g., only on a particular screen of the app). As another example, each navigation button ("back" and "menu") is universally relevant, but it typically has very different behavior on different screens. These observations motivate our final and default selection strategy BiasedRandom.

This strategy is shown in Algorithm 2. Like the Frequency strategy, it maintains a history of how often each event has been selected in the past, but it does so in a context-sensitive manner: the *context* for an event e at a particular instant is the set E of all relevant events at that instant. This history is recorded in global variable G that maps each pair (e, E) to a score. The map starts empty and is populated lazily. At any instant, the score for a pair (e, E) in the map is either -1, meaning event e is blacklisted in context E (i.e., e will never be selected in context E), or it is a non-negative integer, with higher values denoting lesser chance of selecting event e in context E in the future. We found it suitable to use the set of relevant events E as context because it is efficient to compute (the OBSERVER already computes E) and it strikes a good balance between factoring too little and too much of the state into the context.

Each time the SELECTOR is called using this strategy in an observe-select-execute cycle, it runs the algorithm

Algorithm 2 Event selection algorithm BiasedRandom.

```
1: var G : map from (event, set of events) pairs to int
2: G := empty map
3: INPUT: Set E of relevant events.
4: OUTPUT: An event in E.
5: for each (e in E) do
6:   if ((e, E) is not in domain of G) then
7:     // Initialize score of event e in context E.
8:     G(e, E) := init_score(e)
9:   end if
10: end for
11: var L : map from events to int
12: L := map from each event in E to 0
13: while true do
14:   e := event chosen uniformly at random from E
15:   if (L(e) = G(e, E)) then
16:     // Select e this time, but decrease its chance of being
17:     // selected in context E in future calls to SELECTOR.
18:     G(e, E) := G(e, E) + 1
19:     return e
20:   else
21:     // Increase chance of selecting e in next iteration in
22:     // current call to SELECTOR.
23:     L(e) := L(e) + 1
24:   end if
25: end while
26: procedure init_score(e) : int
27:   case (e) of
28:     Text event: return -1
29:     non-Text UI event: return 1
30:     system event: return 2
31:   end case
```

on lines 3-25, taking as input the set of relevant events E from the OBSERVER and producing as output the selected event $e \in E$ for the EXECUTOR to execute. It starts by mapping, for every $e \in E$, the pair (e, E) to its initial score in global map G, unless it already exists in G. We bias the initial score (lines 26-31) depending on the kind of event. If e is a Text event, its initial score is -1. In other words, text inputs are blacklisted in all contexts. Intuitively, the reason is that text inputs are interesting only if they are followed by a non-text input, e.g., a button click. Hence, we forbid selecting text inputs in the SELECTOR altogether, and instead require the EXECUTOR to populate all text boxes in the current UI before it executes the selected non-Text event. We distinguish between two kinds of non-Text events: non-Text UI events and system events. We choose initial score of 1 for the former and 2 for the latter, reducing the relative chance of selecting system events. This bias stems from our observation that system events tend to be relevant over longer periods than UI events, with UI events typically being relevant only when a certain screen is displayed. A hallmark of our algorithm, however, is that it never starves any event in any context. This is ensured by lines 11-25, which repeatedly pick an event e from E uniformly at random, until one satisfies condition $L(e) = G(e, E)$, in which case it is returned as the event selected by the current SELECTOR call. Just before returning, we increment $G(e, E)$ to reduce the chance of picking e in context E in a future SELECTOR call. L is a local map that records the number of times event e was randomly picked by the above process in the current SELECTOR call but passed over for not satisfying the condition. Thus, the lesser the number of times that e has been chosen in context E in past SELECTOR calls, or the higher the number of times that e has been passed over in context E in the current SELECTOR call, the higher the chance that e will be selected to execute in context E in the current SELECTOR call.

Feature Name	Feature Value
Device RAM size	4 GB
Emulator hardware features	All features enabled except GPU emulation
Sdcard size	1 GB
Files on Sdcard (type:count)	pdf:2, img:2, vcf:11, arr:2, zip:4, 3gp:1, m4v:1, mov:1, mp3:3

Table 5: Emulator configuration used in our evaluation.

7. EMPIRICAL EVALUATION

We evaluated the performance of Dynodroid on real-world Android apps, and compared it to two state-of-the-art approaches for testing such apps: manual testing and automated fuzzing. Detailed results of our evaluation are at `http://dynodroid.gatech.edu/study`, including: a VHD file containing Dynodroid's sources and binaries; feedback from a user study we conducted; and a portal allowing users to run Dynodroid on arbitrary provided apps.

All our experiments were done using the Gingerbread version (Android 2.3.5 - API Level 10) which is the most popular version, used by 50% of all devices that recently accessed Google Play [7]. All experiments were done on 64-bit Linux machines with 128GB memory and dual-socket 16-core AMD Opteron 3.0GHz processors. Table 5 shows the emulator configuration (called Android Virtual Device or AVD) that we used in all experiments. For each run, an app was given a freshly created emulator along with only default system applications and the above configuration. After every run, we destroyed the emulator to prevent it from affecting other runs.

We next describe two studies we performed: measuring app source code coverage (Section 7.1) and finding bugs in apps (Section 7.2).

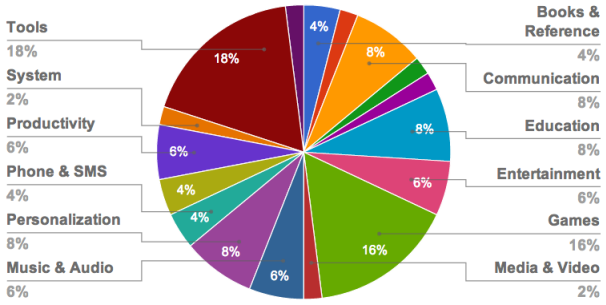


Figure 3: Distribution of open-source apps by category.

7.1 Study 1: App Source Code Coverage

The first study we performed measures the app source code coverage that different input generation approaches are able to achieve. We randomly chose 50 apps from the Android open-source apps repository F-Droid [4] for this study. These 50 apps are sufficiently diverse as evidenced in Figure 3. The SLOC of these apps ranges from 16 to 21.9K, with a mean of 2.7K. The number of components (activities, services, broadcast receivers, and content providers) listed in the AndroidManifest.xml file of these apps ranges from 1 to 38, with a mean of 6.7.

Measuring Coverage. We obtained app coverage metrics by using Emma [3], a popular Java source code coverage tool. Emma generates detailed line coverage metrics to the granularity of branches, and provides coverage reports in different formats that assist in analysis and gathering statistics. By default, it produces coverage reports after the app under test terminates. To generate an intermediate coverage report, a method dumpCoverageData() must be called from the target Java process. As we need to measure code coverage at arbitrary points, a mechanism is needed to inject a method call within the app testing process. To achieve this, we use the Android broadcast intent delivery and broadcast receiver invocation mechanisms, a broadcast receiver with a custom action which calls the above method. To enable Emma, we run each app through an instrumentation activity which starts the main activity of the app. Finally, we change the app's AndroidManifest.xml file by adding the above broadcast receiver and instrumentation activity to it.

Evaluated Approaches. We evaluated the following five approaches in this study: Dynodroid using each of the three selection strategies (Frequency, UniformRandom, BiasedRandom); the Monkey fuzz testing tool provided in the Android platform; and manual testing conducted in a study involving ten users. Table 6 shows the setup we used for each of these five approaches on each app. We ran each of the three variants of Dynodroid for 2,000 events, we ran Monkey for 10,000 events, and we allowed the users in our study to manually generate an unlimited number of events.

Approach	#Events	#Runs
Dynodroid Frequency	2,000	1
Dynodroid UniformRandom	2,000	3
Dynodroid BiasedRandom	2,000	3
Monkey	10,000	3
Humans	no limit	≥ 2

Table 6: Testing approaches used to test each app.

Event Type	Proportion
Touch	15%
Motion	10%
Trackball	15%
Minor Navigation	25%
Major Navigation	15%
System keys	2%
Apps Switch	2%
Others (keyboard, volume, and camera buttons)	16%

Table 7: Kinds of UI events triggered by Monkey.

We used different numbers of events for Dynodroid and Monkey because those are the numbers of events that the two tools were able to generate in roughly the same duration in three hours for each of the 50 apps on average. Dynodroid runs 5X slower than Monkey primarily due to performance issues with the version of the off-the-shelf Hierarchy Viewer tool it calls after each event (see Section 8). On the plus side, as we show below, Dynodroid achieves peak code coverage much faster than Monkey, requiring far fewer than even the 2,000 events we generated.

Monkey triggers a large variety of UI events but no system events. Table 7 summarizes the kinds and proportions of UI events it triggers in its default configuration that we used. The kinds of UI events that Monkey can generate is strictly a superset of those that Dynodroid can generate (see Section 5.1).

All ten users that we chose in our study are graduate students at Georgia Tech who have experience with not only using Android apps, but also developing and testing them using Android developer tools. We provided each of them with each app's source code, the ability to run the app any number of times in the Android emulator, and the ability to inspect app source code coverage reports produced from those runs. They were allowed to provide any kind of GUI inputs, including intelligent game inputs and login credentials to websites. They were also allowed to modify the environment by adding/removing files from the emulator's Sdcard, to manually trigger system events via a terminal by studying the apps' source code, etc.

We ensured that each app was assigned to at least two users. Likewise, we ran each automated approach involving randomization (Monkey, and the UniformRandom and BiasedRandom strategies in Dynodroid) three times on each app. We considered the highest coverage that a user or run achieved for each app. Perhaps surprisingly, for certain apps, we found fairly significant variation in coverage achieved across the three runs by any of the random approaches. Upon closer inspection, we found certain events in these apps that if not selected in a certain state, irreversibly prevent exploring entire parts of the app's state space. Two possible fixes to this problem are: (i) allowing a relatively expensive event during testing that removes and re-installs the app (Dynodroid currently installs the app only once in a run); and (ii) to simply run the app multiple times and aggregate their results.

Finally, Android apps may often call other apps such as a browser, a picture editor, etc. To prevent the automated approaches in the study from wandering far beyond the app under test, we prevented both Dynodroid and Monkey from exercising components not contained in the app under test. In Dynodroid, we achieve this by simply using the "back" navigation button whenever the app under test starts an activity that does not belong to that app. In Monkey, we achieve this by restricting "App Switch" events (see Table 7) to only activities in the app under test.

Coverage Results. The results of our code coverage study for the 50 apps are summarized in the three plots in Figure 4. To enable comparisons for a particular app across plots, each point on the X axis of all three plots denotes the same app. We next elaborate upon the results in each of these plots.

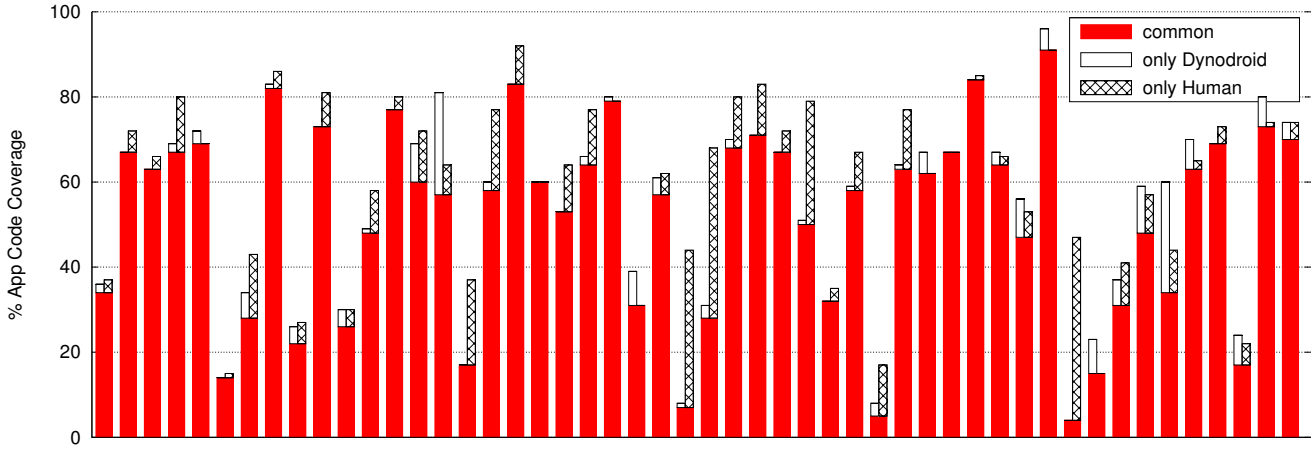
Figure 4a compares the code coverage achieved for each of the 50 apps by Dynodroid vs. Human, where Human denotes the user who achieved the best coverage of all users in our user study for a given app, and Dynodroid uses the BiasedRandom strategy. The bar for each app has three parts: the bottom red part shows the fraction of code that both Dynodroid and Human were able to cover (i.e., the intersection of code covered by them). Atop this part are two parts showing the fraction of code that only Dynodroid and only Human were able to cover (i.e., the code covered by one but not the other).

Both Dynodroid and Human cover 4-91% of code per app, for an average of 51%. Dynodroid exclusively covers 0-26% of code, for an average of 4%, and Human exclusively covers 0-43% of code, for an average of 7%. In terms of the total code covered for each app, Human easily outperforms Dynodroid, achieving higher coverage for 34 of the 50 apps. This is not surprising, given that the users in our study were expert Android users, could provide intelligent text inputs and event sequences, and most importantly, could inspect Emma's coverage reports and attempt to trigger events to cover any missed code.

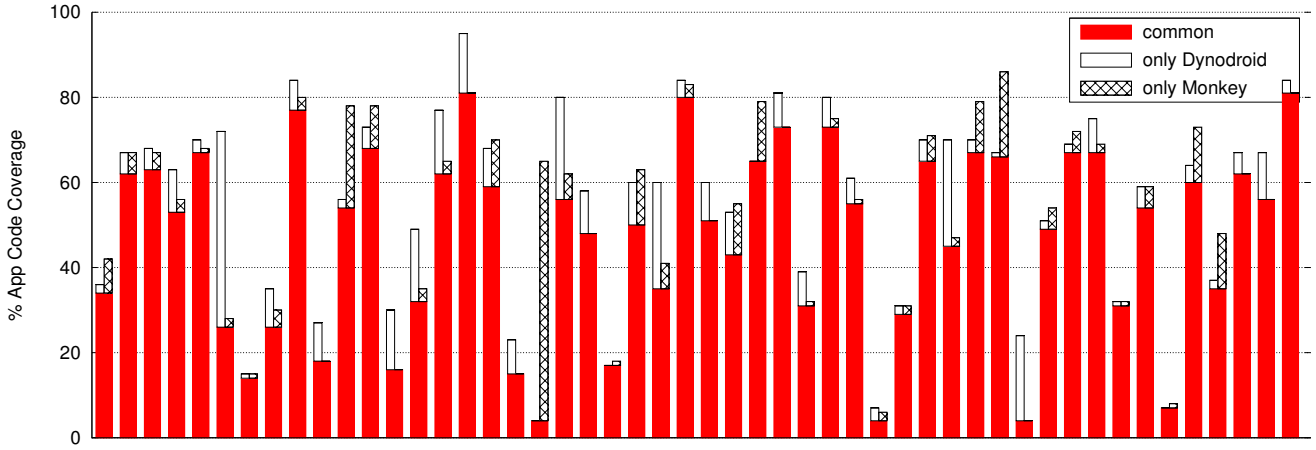
But all the ten users in our study also reported tediousness during testing, how easy it was to miss combinations of events, and that it was especially mundane to click various options in the settings of apps one by one. Dynodroid could be used to automate most of the testing effort of Human, as measured by what we call the *automation degree*, measured as the ratio of coverage achieved by the intersection of Dynodroid and Human, to the total coverage achieved by Human. This ratio varies from 8% to 100% across our 50 apps, with mean 83% and standard deviation 21%. These observations justify Dynodroid's vision of synergistically combine human and machine. It already provides support for intelligent text inputs, where a user with knowledge of an app can specify the text that it should use (instead of random text) in the specific text box widget prior to execution, or can pause its event generation when it reaches the screen, key in the input, and let it resume (as described in Section 4).

Figure 4b compares the code coverage achieved for each of the 50 apps by Dynodroid vs. Monkey. It is analogous to Figure 4a with Monkey instead of Human. Both Dynodroid and Monkey cover 4-81% of code per app, for an average of 47%. Dynodroid exclusively covers 0-46% of code, for an average of 8%, which is attributed to system events that only Dynodroid can trigger. We note that many Android malwares are triggered only when specific system events are triggered [29]; Monkey would not be able to expose such malwares. Monkey exclusively covers 0-61% of code, for an average of 6%, which is attributed to the richer set of UI events that Monkey can trigger (see Table 2 vs. Table 7). Another reason is that Dynodroid only generates straight (Drag) gestures but Monkey combines short sequences of such gestures to generate more complex (e.g., circular) gestures. Finally, Dynodroid uses fixed parameter values for UI events whereas Monkey uses random values, giving it superior ability to exercise custom widgets. In terms of the total code covered for each app, however, Dynodroid outperforms Monkey, achieving higher coverage for 30 of the 50 apps.

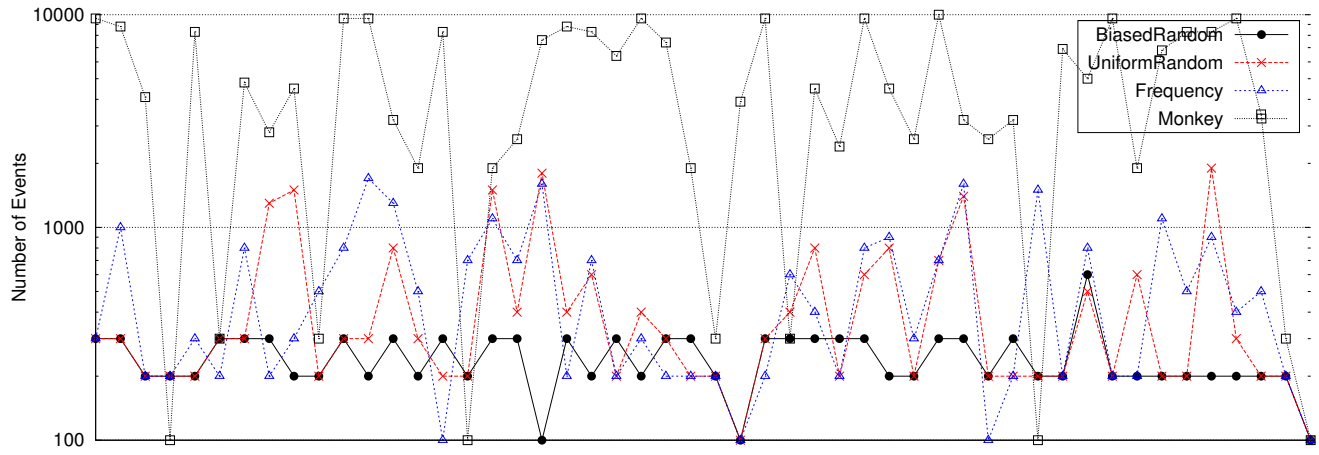
Figure 4c compares the minimum number of events that were needed by each automated approach—Monkey, and Dynodroid using each of the selection strategies—to achieve peak code coverage for each of the 50 apps (recall that we ran Monkey for 10,000 events and Dynodroid for 2,000 events). To strike a good tradeoff between measurement accuracy and performance, we invoke Emma to aggregate coverage after every 100 events for each approach on each app, and hence the minimum number of reported events is 100. It is evident that all three strategies in Dynodroid require significantly fewer



(a) Code coverage achieved by Dynodroid (BiasedRandom) vs. Human.



(b) Code coverage achieved by Dynodroid (BiasedRandom) vs. Monkey.



(c) Minimum number of events needed for peak code coverage by various approaches.

Figure 4: Results of the app source code coverage study. Each point on the X axis in all three plots denotes the same app from the 50 open-source apps used in the study. Figure 4a shows that Dynodroid can be used to automate to a significant degree the tedious testing done by humans. Figure 4b shows that Dynodroid and Monkey get comparable coverage, but Figure 4c shows that Monkey requires significantly more events to do so. Note that the Y axis in Figure 4c uses a logarithmic scale.

App Name	# Bugs	Kind	Description
PasswordMakerProForAndroid	1	NULL_PTR	Improper handling of user data.
com.morphoss.acal	1	NULL_PTR	Dereferencing null returned by an online service.
hu.vsza.adsdroid	2	NULL_PTR	Dereferencing null returned by an online service.
cri.sanity	1	NULL_PTR	Improper handling of user data.
com.zoffcc.applications.aagtl	2	NULL_PTR	Dereferencing null returned by an online service.
org.beide.bomber	1	ARRAY_IDX	Game indexes an array with improper index.
com.addi	1	NULL_PTR	Improper handling of user data.
com.ibm.events.android.usopen	1	NULL_PTR	Null pointer check missed in onCreate() of an activity.
com.nullsoft.winamp	2	NULL_PTR	Improper handling of RSS feeds read from online service.
com.almalence.night	1	NULL_PTR	Null pointer check missed in onCreate() of an activity.
com.avast.android.mobilesecurity	1	NULL_PTR	Receiver callback fails to check for null in optional data.
com.aviary.android.feather	1	NULL_PTR	Receiver callback fails to check for null in optional data.

Table 8: Bugs found by Dynodroid in the 50 open-source apps from F-Droid and the 1,000 top free apps from Google Play. The two classes of apps are separated by the double line, with all the open-source apps listed above. NULL_PTR denotes a "null pointer dereference" exception and ARRAY_IDX an "array index out of bounds" exception.

events than Monkey; in particular, Monkey requires 20X more events than BiasedRandom on average. This is despite Dynodroid considering both system and UI events at each step. The reason is that Dynodroid only exercises relevant events at each step and also because it identifies observationally equivalent events. Finally, of the three selection strategies in Dynodroid, BiasedRandom performs the best, with each of the other two strategies requiring 2X more events than it on average.

7.2 Study 2: Bugs Found in Apps

The second study we performed shows that Dynodroid is an effective bug-finding tool and is also robust. To demonstrate its robustness, we were able to successfully run Dynodroid on the 1,000 most popular free apps from Google Play. The popularity metric used is a score given to each app by Google that depends on various factors like number of downloads and ratings. These apps are uniformly distributed over all 31 app categories in Google Play: the minimum, maximum, mean, and standard deviation of the number of apps in these categories is 26, 55, 40.3, and 6.3, respectively.

We also found that Dynodroid exposed several bugs in both the 50 open-source apps we chose from F-Droid and the 1,000 most popular free apps from Google Play. Table 8 summarizes these bugs. We mined the Android emulator logs for any unhandled exceptions that were thrown from code in packages of the app under test while Dynodroid exercised the app in the emulator. To be conservative, we checked for only FATAL EXCEPTION, as this exception is the most severe and causes the app to be forcefully terminated. We manually ascertained each bug to eliminate any false positives reported by this method but found that all the bugs were indeed genuine and did cause the app to crash.

8. LIMITATIONS

This section outlines the limitations of Dynodroid and suggests ways to overcome them.

Dynodroid needs significantly fewer number of events to converge than Monkey (only 5% of Monkey on average in our experiments) but it is 5X slower than Monkey. The primary reason for the slowdown is that the ViewServer service in Android, which provides the view hierarchy that is used by the OBSERVER in Dynodroid to compute relevant UI events, is slow due to heavy use of reflection. Recent work claims to have patched this problem by introducing a new command DUMPQ that makes ViewServer run 20X-40X faster [1].

Dynodroid restricts apps from communicating with other apps and reverts to the app under test upon observing such communication. However, many Android apps use other apps for shared functionality (e.g., browsing, cropping a picture, etc.). Data is passed between apps via an object called a Bundle which is a key-value store of objects. Dynodroid could be extended with symbolic execution to synthesize relevant Bundle objects. Indeed, the benefits of combining random and symbolic execution have been shown in other domains [20]. A related issue is that Dynodroid uses arbitrary fixed parameter values for most events (e.g., geolocation or touchscreen coordinates) which can prevent it from exercising app code that requires different parameter values. Randomizing and/or symbolically inferring parameter values can address this problem.

Non-determinism in programs is problematic for any dynamic analysis but it is accentuated in Android by the fact that apps use concurrency and asynchrony heavily. One simple way to alleviate non-determinism would be to treat all asynchronous operations synchronously.

Dynodroid currently supports only the Gingerbread version of Android. This may hinder its adoption for

a fast-evolving platform like Android. This problem, however, is mitigated by the fact that Dynodroid instruments the SDK at the Java source level, and a patch could be created using a diff tool and applied to other Android versions without significant effort.

9. CONCLUSION

We presented a practical system Dynodroid for generating relevant inputs to mobile apps on the dominant Android platform. It uses a novel "observe-select-execute" principle to efficiently generate a sequence of such inputs to an app. It operates on unmodified app binaries, it can generate both UI inputs and system inputs, and it allows combining inputs from human and machine. We applied it to a suite of 50 diverse, real-world open-source apps, and compared its performance to two state-of-the-art input generation approaches for Android apps: manual testing done in a user study involving expert Android users, and fuzz testing embodied in the popular Monkey tool provided by the Android platform. We showed that Dynodroid can significantly automate testing tasks that users consider tedious, and generates significantly more concise input sequences than Monkey. We also showed its robustness by applying it to the top 1,000 free apps on Google Play. Lastly, it exposed a few bugs in a handful of the apps to which it was applied.

10. REFERENCES

[1] android-app-testing-patches. `http://code.google.com/p/android-app-testing-patches/`.

[2] DroidBox: Android application sandbox. `http://code.google.com/p/droidbox/`.

[3] EMMA: a free Java code coverage tool. `http://emma.sourceforge.net/`.

[4] Free and Open Source App Repository. `https://f-droid.org/`.

[5] GUITAR: A model-based system for automated GUI testing. `http://guitar.sourceforge.net/`.

[6] Hierarchy Viewer. `http://developer.android.com/tools/help/hierarchy-viewer.html`.

[7] Historical distribution of Android versions in use. `http://developer.android.com/about/dashboards/index.html`.

[8] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`.

[9] D. Amalfitano, A. Fasolino, S. Carmine, A. Memon, and P. Tramontana. Using GUI ripping for automated testing of Android applications. In *Proceedings of 27th Intl. Conf. on Automated Software Engineering (ASE)*, 2012.

[10] S. Anand, M. Naik, H. Yang, and M. Harrold. Automated concolic testing of smartphone apps. In *Proceedings of ACM Conf. on Foundations of Software Engineering (FSE)*, 2012.

[11] R. Bryce, S. Sampath, and A. Memon. Developing a single model and test prioritization strategies for event-driven software. *Trans. on Soft. Engr.*, 37(1), 2011.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2008.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] P. Gilbert, B.-G. Chun, L. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of 2nd Intl. Workshop on Mobile Cloud Computing and Services (MCS)*, 2011.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[16] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proceedings of 6th IEEE/ACM Workshop on Automation of Software Test (AST)*, 2011.

[17] J. Jeon, K. Micinski, and J. Foster. Symdroid: Symbolic execution for dalvik bytecode, 2012. `http://www.cs.umd.edu/~jfoster/papers/symdroid.pdf`.

[18] J. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[19] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In *Proceedings of 7th IEEE/ACM Workshop on Automation of Software Test (AST)*, 2012.

[20] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of 29th Intl. Conf. on Software Engineering (ICSE)*, 2007.

[21] A. Memon, M. Pollack, and M. Soffa. Automated test oracles for GUIs. In *Proceedings of ACM Conf. on Foundations of Software Engineering (FSE)*, 2000.

[22] A. Memon and M. Soffa. Regression testing of GUIs. In *Proceedings of ACM Conf. on Foundations of Software Engineering (FSE)*, 2003.

[23] N. Mirzaei, S. Malek, C. Pasareanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. In *Java Pathfinder Workshop (JPF)*, 2012.

[24] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android app. In *Proceedings of 4th Intl. Conf. on Software Testing, Verification and Validation (ICST)*, 2011.

[25] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of 11th IEEE Intl. Symp. on Software Reliability Engineering (ISSRE)*, 2000.

[26] L. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of 21st USENIX Security Symposium*, 2012.

[27] X. Yuan, M. Cohen, and A. Memon. GUI interaction testing: Incorporating event context. *Trans. on Soft. Engr.*, 37(4), 2011.

[28] X. Yuan and A. Memon. Generating event sequence-based test cases using GUI runtime state feedback. *Trans. on Soft. Engr.*, 36(1), 2010.

[29] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symp. Security and Privacy*, 2012.