# Automating GUI Testing for Android Applications

Cuixiong Hu Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
{huc,neamtiu}@cs.ucr.edu

## ABSTRACT

Users increasingly rely on mobile applications for computational needs. Google Android is a popular mobile platform, hence the reliability of Android applications is becoming increasingly important. Many Android correctness issues, however, fall outside the scope of traditional verification techniques, as they are due to the novelty of the platform and its GUI-oriented application construction paradigm. In this paper we present an approach for automating the testing process for Android applications, with a focus on GUI bugs. We first conduct a bug mining study to understand the nature and frequency of bugs affecting Android applications; our study finds that GUI bugs are quite numerous. Next, we present techniques for detecting GUI bugs by automatic generation of test cases, feeding the application random events, instrumenting the VM, producing log/trace files and analyzing them post-run. We show how these techniques helped to re-discover existing bugs and find new bugs, and how they could be used to prevent certain bug categories. We believe our study and techniques have the potential to help developers increase the quality of Android applications.

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools,Tracing*

## General Terms

Reliability, Verification

## Keywords

Test automation, Mobile applications, Google Android, GUI testing, Test case generation, Empirical bug studies

## 1. INTRODUCTION

Smartphones are becoming pervasive, with more than 195 million sold worldwide in the first three quarters of 2010 alone [23, 22, 24]. A major draw of any smartphone is its ability to run applications, thus users are increasingly relying on smartphones for computing needs, rather than using laptops or desktops. This leads to an increasing impetus for ensuring the reliability of mobile applications. Reliability is particularly important for sensitive mobile applications such as online banking, business management, health care, or military domains.

In this paper we focus on ensuring the reliability of mobile applications running on the Google Android platform. According to Fall 2010 reports, Android is the second most popular mobile OS, surpassing BlackBerry and iPhone OS, and will be tied for number one with Nokia's Symbian by 2014 [29, 24]; Android is in fact the only mobile OS platform to gain market share since Q4'09 [23, 22, 24]. The Android ecosystem includes the Android Market, which currently lists more than 220,000 applications, 12,316 of which were added in December 2010 alone, and an estimated 2.6 billion downloads [17].

Many tools and techniques exist for automating the testing of mature, well-established applications, such as desktop or server programs. However, the physical constraints of mobile devices (e.g., low-power CPU, small memory, small display), as well as developers' unfamiliarity with mobile platforms (due to their novelty), make mobile applications prone to new kinds of bugs. For example, an Android application is structured around activities (GUI windows), broadcast receivers, services and content providers; this is different from standard server applications, or from an event-based system used in a desktop GUI application. The tendency of mobile applications to have bugs is evidenced by their high defect density: a study by Maji et al. [27] has found that Android applications can have defect densities orders of magnitude higher than the OS.

In this paper we aim to bring novel, Android-specific classes of bugs to light, and show how to construct an effective test automation approach for addressing such bugs, especially GUI bugs, and ensuring the reliability of Android applications. First, we conduct a bug collection and categorization on 10 popular open source Android applications (Section 2). We found that, while bugs related to application logic are still present, the remaining bugs are Android-specific, i.e., due to the activity- and event-based nature of Android applications. We categorized all confirmed bugs in the bug database based on our observations. To detect and fix these categories of bugs, we employ an automated test approach (Section 4). Our approach uses a combination of techniques. First, we employ test and event generators to construct test


*AST* '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

cases and sequences of events. We then run these test cases (and feed the events, respectively) to the application. Once a test case is running, we record detailed information about the application in the system log file; after each test case run, we perform a log file analysis to detect potential bugs. To demonstrate the effectiveness of our approach, in Section 5 we present an evaluation on the open source applications that form the object of our bug study. We generated test cases for all projects used in the bug study and compared bugs we found with bugs reported by users. We detected most bugs reported, and found new bugs which have never been reported.

In summary, our work tackles the challenges of verifying mobile applications and makes two contributions:

1. A bug study and categorization of Android-specific bugs that shows an important number of Android bugs manifest themselves in a unique way that is different from traditional, e.g., desktop/server application bugs.

2. An effective approach for detecting Android GUI bugs, based on a combination of test case and event generation with runtime monitoring and log file analysis.

## 2. ANDROID BUGS: A STUDY

To identify the most frequent Android bug categories, we performed an empirical study (bug collection and categorization) on 10 popular applications in the Android Market—the official repository for Android applications. In selecting the applications for our study, we used several criteria: applications had to be popular, have a long lifetime, have a detailed bug history and have the source code available. The 10 applications that form the target of our bug study are available for free in Android Market, have high download counts, and cover most of application categories, which ensures we get a broad range of representative bugs.

The time frame of our analysis, for each application, is shown in columns 2 and 3 in Table 1. As Android OS has only been available for 2.5 years (since August 2008), these applications have had a relatively long lifespan, which has given developers and users a chance to detect and report issues; hence it allows us to observe a wide range of bugs.

We now provide a brief description of each application. Opensudoku [8] is a popular Sudoku game which allows users to download and create their own puzzle in the game; the aim of Opensudoku was to create a framework which can be used for any kind of game, and provide as much functionality and flexibility as possible. Skylight1 [10] is a Java mobile projects framework and collection of Android mobile applications and demos. CMIS [2] is a browser, which enables the user to browse and search CMIS (Content Management Interoperability Services) repositories. Delicious [3] (our abbreviation for Android delicious bookmarks) allows users to save bookmarks to the Delicious social bookmarking service from the Android web browser. ConnectBot [4] is a Secure Shell client, which allows Android users to securely connect to remote servers. DealDroid [5] is a small application for Android devices that continuously watches for new deals on deal sites; it runs in the background and produces Android notifications when new items become available. Rokon [9] is a 2D game engine, intended as a flexible game creation framework with several demo games embedded. Andoku [1] is Sudoku-type puzzle game. MonolithAndroid [7] (recently renamed to Robotic Space Rock) is an OpenGL-based 3D game. GuessTheNumber [6] is a number guessing game.

The results of the study are presented in Table 1. The first column contains the application name, the second column shows the first release we analyzed, and the third column shows the last release we analyzed; source code was collected from Google Code [13] for each application. The rest of the columns (4–11) show the bug counts for each application, categorized by bug type; the bug reports were retrieved from Google Code [13]. The grey-background columns (4–6) are the bugs we focus on in this paper. We now provide a description of each bug type.

*Activities* are the main GUI components of an Android application; an activity error (column 4) usually occurs due to incorrect implementations of the activity protocol, as explained in detail in Section 4.4.1. *Event* errors (column 5) occur when the application performs a wrong action as a result of receiving an event (details in Section 4.4.2). Dynamic *type* errors (column 6) arise from runtime type exceptions (details in Section 4.4.3). *Unhandled exceptions* (column 7) are exceptions the user code does not catch and lead to an application crash. *API* errors (column 8) are caused by incompatibilities between the API version assumed by the application and the API version provided by the system. *I/O* errors (column 9) stem from I/O interaction, e.g., file or card access errors. *Concurrency* errors (column 10) occur due to the interaction of multiple processes or threads. Bugs categorized as *other* (column 11) are due to errors in the program logic.

The bug categories are non-overlapping, i.e., a bug listed in the table cannot belong to more than one category. The only exception to this rule was a bug in ConnectBot 1.46, which can be categorized as both a *type* error and an *API* error (we have categorized it as a type error). As we can see in Table 1, many errors are program logic related errors, some of which can be found using standard techniques such as static analysis or model checking. However, these techniques cannot always be applied directly to mobile applications—their structure and libraries differ substantially from standard applications; also, building or extracting a model for each application could be time-consuming. Therefore, we prefer an automated approach that can detect a variety of bugs, without any per-application effort. We illustrate our approach by showing how it can be used to detect *activity*, *event*, and *type* errors. In the remainder of the paper we present the tools and techniques we used to find such errors, and illustrate how our techniques have re-discovered some known issues, as well as found new bugs.

## 3. ANDROID OVERVIEW

We now proceed to presenting an overview of the Android platform and the components of an Android application. As shown in Figure 1, the Android platform is composed of 4 layers: *Applications* at the top, an *Application Framework* layer that provides services to applications, e.g., controlling activities or providing data access, a *Library/VM* layer, and, at the bottom, the Linux kernel.

Applications run at the very top of the platform. Services for applications, e.g., the *Activity Manager*, which controls activities for each application, or *Content Providers* which load the content provider defined by each application while restricting data accessibility across applications are located in the Application Framework layer. The Library/VM layer

| Application | First analyzed release | Last analyzed release | Bug category | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | *Activity* | *Event* | *Type* | *Unhandled exception* | *API* | *I/O* | *Concurrency* | *Other* |
| Skylight1 | Aug. 2009 | July 2010 | 3 | 2 | 0 | 1 | 0 | 0 | 0 | 4 |
| CMIS | Jan. 2010 | Apr. 2010 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 6 |
| Delicious | Feb. 2009 | June 2010 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| ConnectBot | Aug. 2008 | July 2010 | 2 | 8 | 2 | 5 | 1 | 3 | 1 | 57 |
| DealDroid | Mar. 2009 | May 2009 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 8 |
| Rokon | Sep. 2009 | July 2010 | 0 | 6 | 2 | 3 | 0 | 4 | 0 | 14 |
| Andoku | July 2009 | July 2010 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Opensudoku | Apr. 2009 | July 2010 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 5 |
| GuessTheNumber | Feb. 2009 | Nov. 2009 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| MonolithAndroid | Dec. 2008 | Jan. 2010 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 3 |
| *Total* | | | 8 | 21 | 4 | 11 | 4 | 7 | 1 | 102 |

**Table 1: Android applications: study time frame, bug categories and bug counts.**

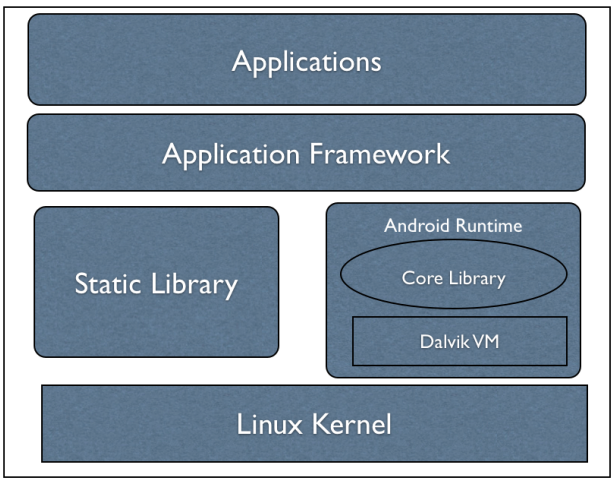


**Figure 1: Architecture of Android platform.**

contains static libraries and the Android runtime environment. Static libraries provide common system and application libraries for applications. The Android runtime environment is composed of core runtime libraries and the Dalvik virtual machine (VM)—an optimized Android-specific Java virtual machine. Finally, the Linux kernel completes the OS and the software stack. Each Android application runs with a unique user ID, in its own copy of the Dalvik virtual machine, which ensures separation between applications and provides protection.

Our work covers the top three layers in Figure 1. To test programs running in the *Application* layer, we use system services from the *Application Framework* layer and instrumentation tools in the *Dalvik VM*.

Android applications can be composed of four component categories: *Activity*, *Broadcast Receiver*, *Content Provider* and *Service*. *Activities* are focused windows in which the user interaction takes place; only one activity can be active at a time. Each activity is a class in the source code and should perform according to events generated by users and system. *Services* run in the background, e.g., an email client may check for new mails while users are running another application. A *Content Provider* manages data for a certain application and controls the accessibility of the data; for example, an email client may make email addresses in its database accessible to other applications. *Broadcast Receivers* listen and react to broadcast announcements. For example, an email client may receive a notification that the battery is low and, as a result, proceed to saving email drafts. Though we believe our approach is general enough to facilitate bug detection for all component classes, in this paper we focus on GUI bugs related to activities and GUI events.

# 4. APPROACH

Our dynamic analysis approach combines several techniques, from automatic test case and event generation tools to log file analysis. In Figure 2 we provide an overview. Starting from an application's source code, we first use JUnit [15], a Java test case generation tool, to generate test cases. Since most applications in the Android Market are GUI-based, for each test case, we may need to add some events (simulating user interaction) to make the application move from one state to another. Therefore, we use Monkey [16], an automatic event generation tool, to produce events in both random and deterministic ways and feed these events to the application. Once a test case is running, we record detailed information about the application in the system log file; after each test case run, we perform a log file analysis to detect potential bugs.

## 4.1 Test Case Generation

JUnit is a testing framework for Java applications, integrated in the Android development environment. JUnit can generate several classes of test cases based on the application source code. Since activities are the main entry points and control flow drivers in Android applications, our test case generation is based on activities. We first identify all activities in an application and then use the Activity Testing class in JUnit to generate test cases for each activity. Activity Testing is shipped with the Android SDK, works in conjunction with JUnit and provides three features:

- *Initial condition testing* tests whether the activity is created properly.

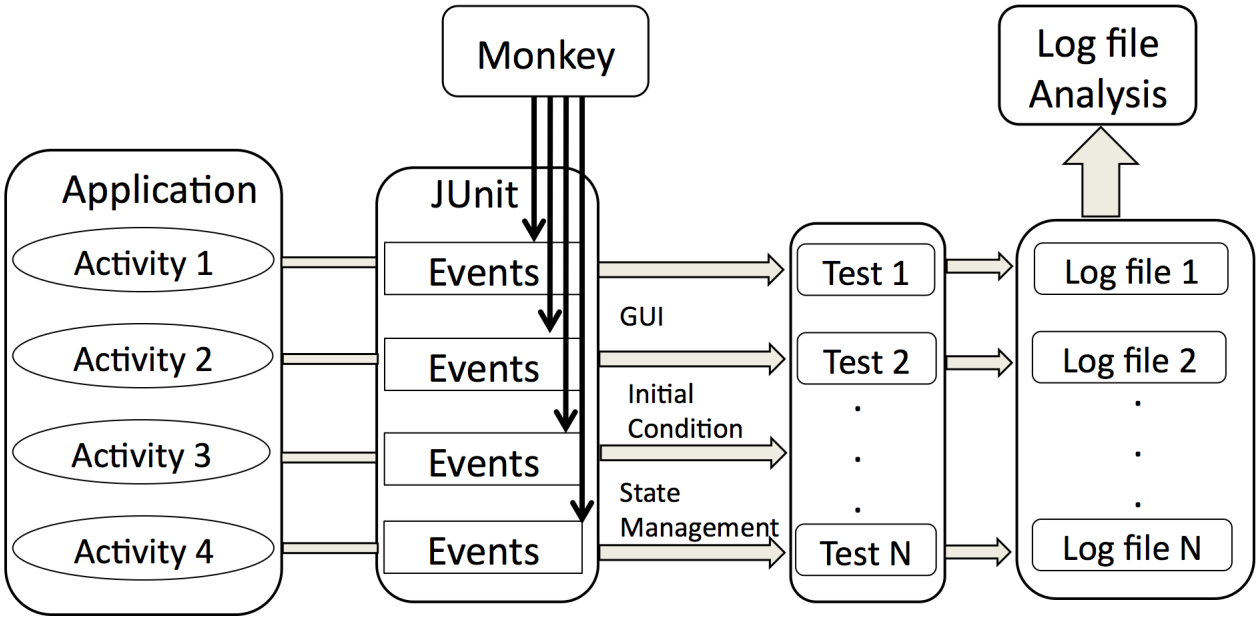


**Figure 2: Overview of our approach.**

- *GUI testing* tests whether the activity performs according to the GUI specification.
- *State management testing* tests whether the application can properly enter and exit a state.

We used all three features for identifying activity bugs. For more effective GUI tests, we used an event generation tool, explained next.

## 4.2 Automatic Event Generation

Automatic event generation is a powerful technique for verifying GUI applications. GUI bugs are revealed by event sequences that fall outside the set of permissible events associated with the current state of the GUI application. To help generate GUI events, we use the Monkey event generator, which comes with the Android SDK. Monkey can generate random or deterministic event sequences and feed these events to an application. To discover a wide range of issues, in our work we use random sequences: we generate these sequences using Monkey, and feed the sequences to the application under test.

## 4.3 Trace Generation

Once the test cases are generated, we run them on the application through the Dalvik VM. To monitor the execution of test cases, we configure the VM to log the details of each test case into a trace file. Our traces capture three kinds of events: GUI events, method calls, and exceptions. We also monitor the VM operation to detect application bugs that cause the VM to shut down prematurely.

## 4.4 Log File Analysis and Bug Detection

Once test cases are generated for a certain application, we run the application on these test cases and log the performance of each test case so that we can detect errors. With the log file at hand, we use patterns to identify potential bugs. Each class of errors (activity, event or type) has an associated "pattern," as explained next. These patterns can indicate proper operation, or they can indicate a bug. Apart from automatic bug detection, log files are also useful in debugging—since the log file contains method and event traces, leading to the bug, developers can use our framework to reconstruct the method sequence that lead to a bug.

### *4.4.1 Detecting Activity Bugs*

Activities are window containers derived from an Activity superclass; their implementations consist of responding to events generated by users and the system. Activity bugs stem from incorrect implementation of the Activity class, e.g., one activity might be created or destroyed in the wrong way so that it will make the application crash. In general, activity bugs occur either because developers are not sufficiently familiar with the activity- and event-based application model in Android, or because the implementation fails to obey the activity state machine. In practice, almost every application we analyzed has activity bugs because it is hard to check whether each base function of the base class has been properly implemented.

An activity has a life cycle described by a state machine, hence violations of this state machine lead to activity bugs. A simplified version of the state machine is shown in Figure 3; the full state machine can be found on the Android developer website [12]. Each activity can be in one of five states: *Active*, *Pause*, *Stop*, *Restore* or *Destroy*. If an activity occupies the screen's foreground, it is running, hence in the *Active* state. If another non-full screen or transparent activity overlaps the current activity, the current activity will be moved into the *Pause* state. An activity is in state *Stop* once it is fully covered by another activity. Activities in states *Stop* or *Pause* can be killed by system if memory is needed elsewhere. If the activity is killed and the user has restarted it again after some time, that activity will be in

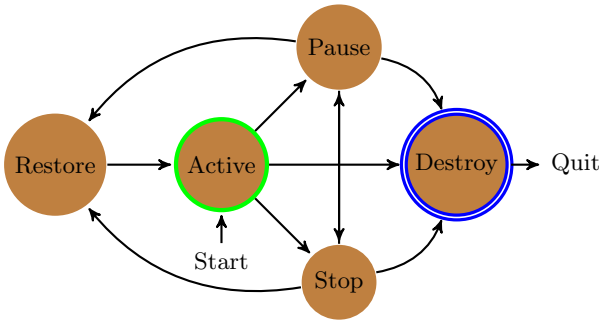


**Figure 3: Simplified state machine of an Android activity.**


**Figure 4: Screenshot of ConnectBot activity failure.**

state *Restore* and then *Active*. Once an activity needs to be killed, it will be in the *Destroy* state.

To ensure a correct state sequence, e.g., Start→ Active→ Pause→ Restore→ Active→ Destroy, the corresponding user-defined activity methods should be called in a valid order as specified by the state machine, in this case: onCreate()→ onPause()→ onResume()→ onDestroy(). We use the state machine as a specification and match method calls from log file entries against it. Violations of the state machine are then flagged as potential bugs.

For example, in ConnectBot release 256 we found a new activity bug, indicated by the log file entries shown in Figure 5 (a). The bug in Figure 5(a) manifests itself as an onCreate() on line 1 without a subsequent onPause() preceding line 3, which is a violation of the state machine specification. The bug corresponds to a situation where the user sets up a default shell host beforehand and then starts the application, which crashes the application. Figure 4 is a screen shot of the application crash when the scenario described above unfolds.

| Program | Activity bugs | | Event bugs | | Type errors | |
|---|---|---|---|---|---|---|
| | *Old* | *New* | *Old* | *New* | *Old* | *New* |
| Skylight1 | 3 | 0 | 2 | 3 | 0 | 0 |
| CMIS | 0 | 0 | 0 | 0 | 0 | 0 |
| Delicious | 0 | 0 | 0 | 0 | 0 | 0 |
| ConnectBot | 2 | 2 | 6 | 2 | 2 | 0 |
| DealDroid | 1 | 0 | 0 | 0 | 0 | 0 |
| Rokon | 0 | 0 | 6 | 0 | 2 | 0 |
| Andoku | 0 | 1 | 0 | 0 | 0 | 0 |
| Opensudoku | 1 | 0 | 1 | 1 | 0 | 0 |
| GuessTheNumber | 1 | 0 | 1 | 0 | 0 | 0 |
| MonolithAndroid | 0 | 0 | 2 | 0 | 0 | 0 |
| *Total* | 8 | 3 | 18 | 6 | 4 | 0 |

**Table 2: Results: old (re-discovered) bugs and new (not previously reported) bugs.**

#### 4.4.2 *Detecting Event Bugs*

Android applications should be prepared to receive events, and react to events, in any state of an activity. If developers fail to provide proper implementations of event handlers associated with certain states, the application can either enter an incorrect state or crash outright. Figure 5(b) is a log file example extracted from ConnectBot release 80, which shows how the application crashes when Monkey feeds it with an unhandled event (HOME button click).

#### 4.4.3 *Detecting Type Errors*

Detecting type errors is quite simple: once the type error has been triggered, a ClassCastException entry will appear in the log file. A type error in ConnectBot release 236 is shown in Figure 5(c).

## 5. RESULTS

Our verification techniques turned out to be effective in practice. In Table 2 we report the number of bugs we found using our approach. For each class of bugs we were able to re-discover bugs already reported (the *Old* columns) as well as new bugs that have not been reported yet (the *New* columns). For assurance, we took each event sequence our approach has found automatically and played it manually, through GUI interaction, to make sure the bug can actually be reproduced in practice. We have reported the new bugs to the developers; two of the bugs have been confirmed, while others are in the process of confirmation.

### 5.1 Activity Bugs

We were able to detect all 8 activity errors that have already been reported (Table 1), as well as 3 new activity errors in Andoku and Connectbot.

### 5.2 Event Bugs

Our technique has detected 24 event errors (18 existing bugs and 6 new bugs). However, Table 1 contains 21 event errors; the three errors we could not detect could not be reproduced by other users either, so we suspect they might be spurious bug reports.

**(a) Activity bug** in ConnectBot release 256

```
E/AndroidRuntime( 190): at org.connectbot. SettingsActivity .onCreate( SettingsActivity .java:29)
E/AndroidRuntime( 190): at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2364)
I/ActivityManager( 52): Process org.connectbot (pid 190) has died.
D/AndroidRuntime( 211): Shutting down VM W/dalvikvm( 211):
threadid=3: thread exiting with uncaught exception (group=0x4001aa28)
E/AndroidRuntime( 211): Uncaught handler: thread main exiting due to uncaught exception
```

**(b) Event bug** in ConnectBot release 80

```
I/Starting activity : Intent{action=android.intent.category.HOME}
...
D/:Shutting down VM
```

**(c) Type error** in ConnectBot release 236

```
D/PackageParser( 63): Scanning package: /data/app/vmdl57744.tmp W/Resources(
63): Converting to int: TypedValue{t=0x3/d=0x11 "Donut" a=1} W/PackageParser(
63): /data/app/vmdl57744.tmp W/PackageParser( 63):
java.lang.NumberFormatException: unable to parse 'Donut' as integer
...
W/PackageParser( 63): at android.os.HandlerThread.run(HandlerThread.java:60)
E/AndroidRuntime( 1555):java.lang.RuntimeException:java.lang.ClassCastException: java.lang.Long;
D/AndroidRuntime: Shutting down VM
```

**Figure 5: Bug manifestation: examples of log file entries for each bug category.**

## 5.3 Type Errors

We successfully detected all type errors in Table 1, but we could not find any new ones—we suspect that, since type errors are critical and cause the applications to crash immediately, they are more likely to be observed, reported and fixed quickly.

# 6. FUTURE WORK

The last column of Table 1 shows bugs that do not fall into the activity/event/type categories. Some examples include *unhandled exceptions*, *API errors*, *I/O errors*, or *concurrency errors*. We plan to pursue a two-prong approach for detecting these classes of bugs. First, we will model the correct invocation of certain I/O and concurrency primitives as state machines [21], which will allow us to compare application logs with the model and find I/O and concurrency errors. Second, we plan to graft our model-based verification onto Java static analysis tools, e.g., WALA [11], to permit pattern and state-machine violations at compile-time, rather than via automated testing.

# 7. RELATED WORK

*Android verification.* Most of the prior work on verification of mobile applications has focused on security. Enck, Ongtang, and McDaniel's [20] describe Kirin, a logic-based tool for Android that ensures permissions needed by a certain Android application are met by global safety invariants. These invariants ensure that data flow of each application can be passed safely among applications. Ongtang, McLaughlin, Enck and McDaniel [28] have proposed a similar tool called Saint, which enforces OS-level data flow security. Enck, McDaniel, Jung and Chun [19] provide a real-time data flow monitor called TaintDroid, which can check data misuse during runtime for an Android application. In their work they tested 30 popular applications and found several cases of information leak, e.g., leaking user contacts information. Chaudhuri [18] presented a formal study of Android security. Their work introduced a core typed system for describing Android applications, and reasoning about their data flow security properties. Any violation of rules described in the core typed system and operational semantics may result in security concerns of applications. These approaches are deployed at the OS level and focus on security, whereas we focus on providing developers with a toolset for detecting GUI errors.

*GUI testing.* Kervinen et al. [26] present a formal model and architecture for testing concurrently running applications. The behavior of the system under test is specified as a labeled transition system. Their specification model is more rigorous and powerful than ours, as our model is sequential, rather than concurrent. They used their system to test mobile applications running on the Symbian platform and found 6 bugs in those applications.

GUITAR [14] is a GUI testing framework for Java and Microsoft Windows applications. Hackner and Memon [25] describe the architecture and overview of the GUITAR tool—the components of GUITAR and their functionality. It is unclear to us whether GUITAR can be applied directly to Android, because of Android's application development model.

Yuan and Memon [30] generate event-sequence based test cases for GUI applications. They proposed a model-based approach for testing GUI-based applications. Their technique can generate test cases automatically using a structural event generation graph. Their approaches target Java desktop applications, which are quite different from the Android mobile environment.

*Android bug studies.* Maji et al. [27] performed a failure characterization study on two mobile platforms, Android and Symbian. They collected data on bugs in the OS, middleware/library, and development tools/core applications for these two platforms. They found that defect density tends to be lowest in the OS, higher in the middleware, and highest in development tools/core applications. They also computed the cyclomatic complexity for the two code bases and found the values to be comparable across all layers. They studied the code fixes and found that many of the fixes required just a few lines of code. Our study is centered around a complementary set of applications (third-party, non-core), since this is the type of applications prevalent on the Android Market. Their study is focused on bug location (which subsystem contains the bug) as well as the fix (what source code change was made to fix the bug), whereas our bug categorization looks at the semantic nature of the bug (e.g., activity, event, type error, concurrency, etc.). Our approach is oriented on dynamic, rather than static verification. We do not study fixes and source code, but rather provide an automatic testing framework that can be used to discover activity bugs, event bugs and type errors.

# 8. CONCLUSIONS

The number of mobile applications and mobile application users are growing rapidly, which creates an impetus for researchers and developers to come up with effective verification techniques to ensure the reliability of these applications. Towards this goal, we perform a bug study to understand the nature and possible remedies for bugs in mobile applications, and construct an automated testing framework for Android applications. Our framework combines automatic event and test case generation with runtime monitoring and log file analysis. Our techniques have proved effective for activity, event, and type errors: we have been able to re-discover existing bugs while finding some new bugs. We believe our framework can be easily extended to find a broader range of bugs in Android applications.

# 9. REFERENCES


[1] andoku. http://code.google.com/p/andoku/.

[2] android-cmis-browser. http://code.google.com/p/android-cmis-browser/.

[3] android-delicious-bookmarks. http://code.google.com/p/android-delicious-bookmarks/.

[4] connectbot. http://code.google.com/p/connectbot/.

[5] dealdroid. http://code.google.com/p/dealdroid/.

[6] guessthenumber. http://code.google.com/p/guessthenumber/.

[7] monolithandroid. http://code.google.com/p/monolithandroid/.

[8] opensudoku-android. http://code.google.com/p/opensudoku-android/.

[9] rokon. http://code.google.com/p/rokon/.

[10] skylight1. http://code.google.com/p/skylight1/.

[11] T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page.

[12] Android activity lifecycle, May 2010. http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle.

[13] Google code, April 2010. http://code.google.com/.

[14] Guitar – a gui testing framework, August 2010. http://guitar.sourceforge.net/index.shtml.

[15] JUnit, May 2010. http://www.junit.org/.

[16] Monkey UI/Application Exerciser, May 2010. http://developer.android.com/guide/developing/tools/monkey.html.

[17] Androlib. Number of New Applications in Android Market by month, September 2010. http://www.androlib.com/appstats.aspx.

[18] A. Chaudhuri. Language-based security on android. In *PLAS '09*, pages 1–7.

[19] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. 2010.

[20] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7(1):50 –57, Jan.-Feb. 2009.

[21] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. SOSP, pages 237–252, 2003.

[22] Gartner Corporation. Gartner Says Worldwide Mobile Device Sales Grew 13.8 Percent in Second Quarter of 2010, But Competition Drove Prices Down, August 2010. http://www.gartner.com/it/page.jsp?id=1421013.

[23] Gartner Corporation. Gartner Says Worldwide Mobile Phone Sales Grew 17 Per Cent in First Quarter 2010, May 2010. http://www.gartner.com/it/page.jsp?id=1372013.

[24] Gartner Corporation. Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010; Smartphone Sales Increased 96 Percent, November 2010. http://www.gartner.com/it/page.jsp?id=1466313.

[25] D. R. Hackner and A. M. Memon. Test case generator for guitar. In *ICSE Companion '08*, pages 959–960.

[26] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. Model-based testing through a gui. In *Formal Approaches to Software Testing*, volume 3997, pages 16–31. 2006.

[27] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 249–258.

[28] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel. Semantically rich application-centric security in android. In *ACSAC'09: Annual Computer Security Applications Conference*, 2009.

[29] M. Swift. Android operating system is expected to surge past rivals. *Los Angeles Times*, Sep 11 2010.

[30] X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. on Software Engineering*, 36:81–95, 2010.