Mutation Operators for Testing Android Apps

Lin Deng*, Jeff Offutt, Paul Ammann, Nariman Mirzaei

Department of Computer Science
George Mason University
Fairfax, Virginia, USA
{*ldeng2, offutt, pammann, nmirzaei*}*@gmu.edu*

Abstract

Context: Due to the widespread use of Android devices, Android applications (*apps*) have more releases, purchases, and downloads than apps for any other mobile devices. The sheer volume of code in these apps creates significant concerns about the quality of the software. However, testing Android apps is different from testing traditional Java programs due to the unique program structure and new features of apps. Simple testing coverage criteria such as statement coverage are insufficient to assure high quality of Android apps. While researchers show significant interest in finding better Android testing approaches, there is still a lack of effective and usable techniques to evaluate their proposed test selection strategies, and to ensure a reasonable number of effective tests.

Objective: As mutation analysis has been found to be an effective way to design tests in other software domains, we hypothesize that it is also a viable solution for Android apps. **Method:** This paper proposes an innovative mutation analysis approach that is specific for Android apps. We define mutation operators specific to the characteristics of Android apps, such as the extensive use of XML files to specify layout and behavior, the inherent event-driven nature, and the unique Activity lifecycle structure. We also report on an empirical study to evaluate these mutation operators. **Results:** We have built a tool that uses the novel Android mutation operators to mutate the source code of Android apps, then generates mutants that can be installed and run on Android devices. We evaluated the effectiveness of Android mutation testing through an empirical study on real-world apps. This

---

*Corresponding author.

Preprint submitted to Information and Software Technology *April 20, 2016*

paper introduces several novel mutation operators based on a fault study of Android apps, presents a significant empirical study with real-world apps, and provides conclusions based on an analysis of the results. **Conclusion:** The results show that the novel Android mutation operators provide comprehensive testing for Android apps. Additionally, as applying mutation testing to Android apps is still at a preliminary stage, we identify challenges, possibilities, and future research directions to make mutation analysis for mobile apps more effective and efficient.

Keywords: Android, Software Testing, Mutation Testing

1. Introduction

A *mobile application* is a software program that runs on a mobile device such as a smartphone or a tablet. The number of mobile applications (*apps*) is growing as more platforms become available, more apps are marketed, prices drop, and more users acquire more devices. The Android operating system currently dominates the market with 83.1% of sales in the third quarter of 2014 (iOS was second with 12.7%) [1]. Over a million apps are available to Android users on the Google Play store, the most widely used Android app store [2], and thousands are added every day.

Not surprisingly, quality is a serious and growing problem. Many apps reach the market containing significant faults, which often result in failures during use. To investigate the pervasiveness of software faults in Android apps, Bhattacharya et al. [3] analyzed 29,233 bug reports in 24 widely-used open source Android apps and found that more than 8,500 of the bug reports were confirmed as faults and fixed by developers later. None of the Android apps they analyzed was fault-free. Although part of the problem is a lack of software engineering (including little or no testing), a significant technical problem also exists. Android apps involve several new programming features and we have very little knowledge about how to test them. This results in weak and ineffective testing. In fact, even among developers who attempt to test their apps well, random value generation is quite common [4]. Although several researchers have proposed improved test techniques [4, 5, 6, 7, 8], these have not reached practice.

The goal of our overall research project is to develop testing techniques that can allow developers to find faults in Android apps before release, especially in the parts of the code that use new programming features (as described in Section 2). Specifically, we propose to use mutation analysis, a high-end testing technique that is known for helping engineers design powerful tests.

We start by analyzing the unique technical features of Android apps, and design novel mutation operators for those features. Tests that kill those mutants can be expected to reveal many faults in the use of the features. We have built a proof-of-concept mutation analysis tool that implements the new Android mutation operators as well as more traditional mutation operators.

Our Android mutation analysis tool can be used in three different ways. As a method for test case design, mutation analysis is one of the most powerful test criteria known. Thus mutation can be used to design very powerful tests. Second, once completed, polished, and made available to other researchers, a mutation analysis tool can be used to evaluate other test techniques for Android apps. Third, if a tester has a large number of pre-existing tests, many are likely to be redundant. This is particularly troublesome for Android testers, because for a variety of technical reasons, test execution tends to be quite slow. However, identifying which tests to keep and which tests to dispose of is a challenging problem. Mutation analysis allows tests to be *filtered* by keeping only tests that increase the mutation score.

The paper makes the following contributions:

- It defines novel mutation operators specific to Android apps.
- It evaluates these mutation operators on eight Android apps.
- It identifies future research areas for mutation analysis of Android apps.

This paper extends our work published at the 2015 Mutation Workshop [9]. The previous experiment revealed some shortcomings, so we have designed new Android mutation operators based on common faults in Android apps, conducted additional empirical studies with new real-world apps, collected results and carried out a thorough analysis, and compared executions between emulators and real devices, as well as different runtime systems. This paper also studies eight apps, compared with only one in the previous paper [9].

This paper is organized as follows. Section 2 describes how Android apps are programmed, including some of the unique aspects of programming in the framework, and introduces how mutation analysis works. Section 3 defines eleven novel mutation operators that mutate new programming features such as the *Intent* and *event handlers.* Section 4 outlines how mutation analysis is applied in the Android framework, which is quite different from traditional languages such as Java. Section 5 presents the Android apps we study, shows how mutation analysis can be used to test them, and describes results for the empirical study. The paper concludes with an overview of the related research in Section 6, and a discussion of our planned future work in Section 7.

2. Background

Android apps are built differently from traditional software, and use new structures and new control and data connections. This research project is applying an existing testing technique, mutation testing, to a new type of software, mobile apps. So before going into our research, we need to provide a brief overview of how Android app works, followed by an overview of mutation testing.

2.1. Programming Android Applications

Android comes with a development environment called the Android Application Development Framework (ADF). Android ADF provides an API to help build apps, create GUIs, and access data on devices. Android includes an operating system based on Linux, including middleware, pre-installed applications, and system libraries. Android used the Dalvik Virtual Machine [10] to execute Java programs before the version of 4.4 (KitKat). The most recent release, Android 5.0 (Lollipop), replaced Dalvik with Android Runtime (ART). However, as stated by Google, most apps developed for Dalvik should work without any changes under ART [11]. The change does not affect the general structure or programming methodology of Android apps. Android apps can also *publish* their features for other apps to use, subject to certain constraints.

Android apps are built according to a novel structure with a mandatory *manifest* file and four types of components. Manifest files are written in eXtensible Markup Language (XML) and provide information about the app to the ADF, including configuration information and descriptions of the apps' components.

Android apps have four types of components: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers.* An *Activity* presents a screen to the

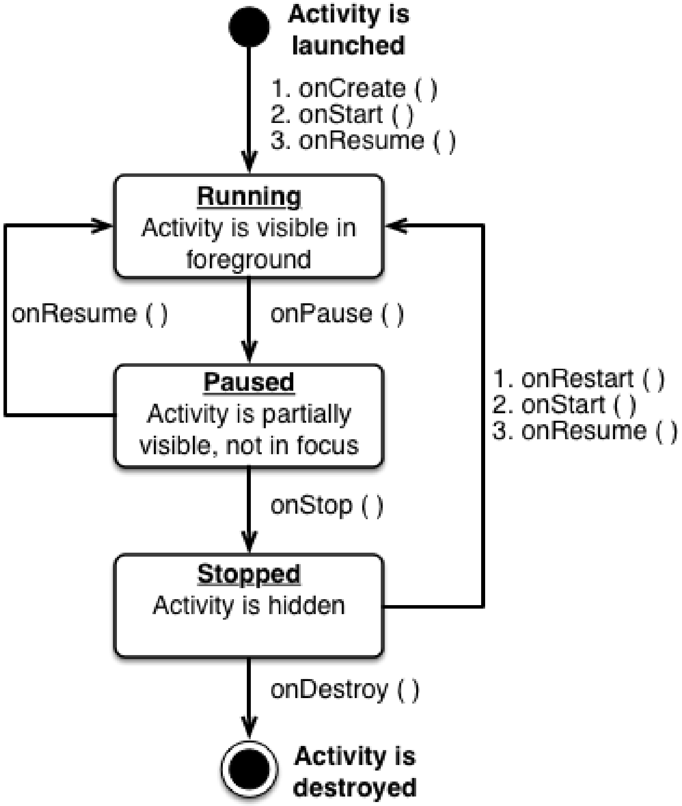


Figure 1: Activity lifecycle in Android apps

user based on one or more layout designs. These layouts can include different configurations for different sized screens. The layouts define *view widgets*, which are GUI controls. A configuration file in XML describes the controls and how they are laid out with a unique identifier for each widget. *Service* components run on the device in the background. They perform tasks that do not require interaction with the user such as counting steps, monitoring set alarms, and playing music. Services do not interact with the screen, although they may interact with an Activity, which in turn interacts with the screen. A *Content Provider* stores and provides access to structured data stored in the file system, including calendar, photographs, contacts, and stored music. Finally, a *Broadcast Receiver* handles messages that are announced system-wide such as low battery.

An Android component is activated by using an *Intent* message, which includes an action that the component should carry out, and data that the component needs. Android supports run-time binding of *Intent* messages. This is enabled by having calls go through the Android messaging service, rather than being explicitly present in the app.

Android requires all major components such as Services and Activities to behave according to a pre-specified lifecycle [12]. The ADF manages these behaviors. Figure 1 shows the lifecycle of an Activity as a collection of events and states. The states are *Running*, *Paused*, and *Stopped*. The *Running* state is reached after events *onCreate()*, *onStart()*, and *onResume()*. *onPause()* sends the Activity to the *Paused* state, then *onStop()* sends it to *Stopped* and *onResume()* sends it back to *Running*. From *Stopped*, the Activity can go to *Running* with *onRestart()*, *onStart()*, or *onResume()*, or it can exit with an *onDestroy()* event. ADF calls lifecycle event handlers and are integral to our research, as explained later.

2.2. Mutation Analysis

This paper proposes the use of mutation to design effective tests for Android app components. Mutation testing modifies a software artifact such as a program, requirements specification, or a configuration file, to create new versions called *mutants* [13]. The mutants are usually intended to be faulty versions and are created by applying rules for changing the syntax of the software artifact. These rules are called *mutation operators*. The tester then creates tests that cause the original and each mutated version to exhibit different behaviors, called *killing* the mutant. For example, the *ROR* operator for traditional programming languages replaces every instance of every relational operator (for example, <=) with all other relational operators (<, ==, >, >=, ! =) plus *trueOp* and *falseOp*, which set the condition to true and false [14]. Mutation operators sometimes create changes that are similar to programmer mistakes, and sometimes introduce changes that force testers to design test inputs that are likely to find faults.

Each mutant is run against the tests in a test suite to measure the percentage of mutants the tests kill. This is called the *mutation adequacy score*. Mutation testing has consistently been found to usually be stronger than other test criteria. One source of that strength is that it does more than just apply local requirements, such as reach a statement or tour a subpath in the control flow graph (*reachability*), but it also requires that the mutated statement result in an error in the program's execution state (*infection*), and that erroneous state propagate to incorrect external behavior of the mutated program (*propagation*) [14, 15, 16].

Some mutants have the same behavior as the original program on every input, so cannot be killed. These mutants are called *equivalent*. Identifying and eliminating equivalent mutants from consideration is a major cost of mutation testing. Some mutants do not compile and become *stillborn* [14, 17, 18, 19] because the change makes the program syntactically incorrect. While these *stillborn* mutants can usually be avoided if the mutation operators are well designed and properly implemented, some do occur. A mutation system must be prepared to recognize stillborn mutants and remove them from consideration.

Mutation operators have been created for many different languages, including C, Java, and Fortran [20, 21, 22, 23]. Mutation operators for Android apps focus on the novel features of Android, including the manifest file, activities, and services.

3. Android Mutation Operators

Mutation analysis relies on mutation operators, which are syntactic rules for changing the program or artifact. Good mutation operators can lead to very effective tests, but poor mutation operators can lead to ineffective tests or large numbers of redundant tests. Mutation operators are usually defined using one of two approaches. When available, mutation operators are defined from fault models where each type of fault is used to design a mutation operator that creates instances of those faults. The muJava class-level operators [24, 25] were based on a previous fault model by Alexander [26]. Another approach is to analyze every syntactic element of the language being mutated, and design mutants to modify the syntax in ways that typical programmers might make mistakes.

We have defined five categories of mutation operators, four of which are based on the Android app elements they cover (Intent, Activity lifecycle, event handler, and XML). The fifth is based on common faults that app programmers make. Every Android app must have at least one Activity [27], making it crucial for testing.

Google's "Activity Testing: What To Test" [28] document lists *Intent* and *lifecycle events* as two essential elements to test. Android apps are event-driven GUI programs [8, 29, 30], thus we designed operators that mutate event handlers. Because aspects of Android apps are defined in XML

Original Type	**Default Value**
int, short, long, float, double, char	0
boolean	true / false
String	“” / (String) null
Array	(Array) null
Others	(Others) null

Table 1: IPR default values

configuration files, we also designed operators to mutate the XML files.

Android apps often have faults based on null values and the orientation of the screen [31]. Thus, we have designed two mutation operators based on those faults.

We have designed eleven mutation operators within these five categories. The following subsections define each operator in turn, organized by the categories.

3.1. Intent Mutation Operators

As described in Section 2, an Intent is an abstraction of an operation to be performed among Android components [32]. They are usually used to launch an activity or transmit data or messages between activities.

*3.1.1. Intent Payload Replacement (**IPR**)*

An Intent can carry different types of data (called payload) as key-value pairs. The *putExtra()* method takes the key name as the first parameter, and the value as the second parameter. The IPR operator mutates the second parameter to a default value that depends on the underlying data type. These default values are listed in Table 1. Objects with primitive numeric types, such as int, short, long, etc., are replaced by the value zero, and boolean variables are replaced by both true and false. String objects are replaced by empty strings and null values. Array and other types of objects are replaced by null values cast into the appropriate types.

Figure 2 shows an example IPR mutant. The String object *message* is replaced with an empty String (the original and mutated statements are in bold face). IPR mutants challenge testers to design test cases to ensure the value passed by an Intent object is correct.

```
public void test (View view)
{
   Intent intent = new Intent (this, DisplayMessageActivity.class);
   EditText editText = (EditText) findViewById (R.id.edit_message);
   String message = editText.getText().toString();
   intent.putExtra (EXTRA_MESSAGE, message);
   startActivity (intent);
}
```

Original

```
public void test (View view)
{
   Intent intent = new Intent (this, DisplayMessageActivity.class);
   EditText editText = (EditText) findViewById (R.id.edit_message);
   String message = editText.getText().toString();
   intent.putExtra (EXTRA_MESSAGE, “”);
   startActivity (intent);
}
```

Mutant

Figure 2: Intent Payload Replacement mutant example

*3.1.2. Intent Target Replacement (**ITR**)*

Developers use an *explicit Intent* to specify which component should be started by declaring the Intent with the target component's name within an app.

Figure 3 shows an Intent object that is declared with *ActivityB.class* as the target. The ITR operator first looks up all the classes within the same package of the current class, and then replaces the target of each Intent with all possible classes. This challenges the tester to design test cases that check that the target activity or service is launched successfully after the Intent is executed.

3.2. Activity Lifecycle Mutation Operator

Section 2 described the pre-specified lifecycle followed by major components, as illustrated in Figure 1. Components use seven methods to fulfill transitions among different states in the lifecycle. This operator modifies those methods.

*3.2.1. Lifecycle Method Deletion (**MDL**)*

Developers override transition methods to define transitions among states. MDL deletes each overriding method to force Android to call the version in

```
public void startActivityB (View v)
{
  Intent intent = new Intent (ActivityA.this, ActivityB.class);
  startActivity (intent);
}
```

Original

```
public void startActivityB (View v)
{
  Intent intent = new Intent (ActivityA.this, ActivityC.class);
  startActivity (intent);
}
```

Mutant

Figure 3: Intent Target Replacement mutant example

the super class. This requires the tester to design tests that ensure the app is in the correct expected state. The MDL operator is similar to the Overriding Method Deletion mutation operator (IOD) in muJava [25], but only considers the methods related to the Activity lifecycle.

3.3. Event Handler Mutation Operators

Android apps are event-based, so event handlers are normally used to recognize and respond to events. Common user actions are clicking and touching, each of which generates an event. Thus, we define two mutation operators for event handlers, the OnClick Event Replacement (ECR) operator, and the OnTouch Event Replacement (ETR) operator.

*3.3.1. OnClick Event Replacement (**ECR**)*

ECR first searches and stores all event handlers that respond to OnClick events in the current class. Then, it replaces each handler with every other compatible handler. Figure 4 shows an ECR mutant where the event handler for the button mPrepUp has been replaced by the event handler for the button mPrepDown. To kill ECR mutants, each widget's OnClick event has to be executed by at least one test.

*3.3.2. OnTouch Event Replacement (**ETR**)*

This operator replaces the event handlers for each OnTouch event. It works exactly the same as the ECR mutation operator.

```
mPrepUp.setOnClickListener (new OnClickListener()
{
   public void onClick (View v) {
      incrementPrepTime();
   }
});
mPrepDown.setOnClickListener (new OnClickListener()
{
   public void onClick (View v) {
      decrementPrepTime();
   }
});
```

Original

```
mPrepUp.setOnClickListener (new OnClickListener()
{
   public void onClick (View v) {
      decrementPrepTime();
   }
});
mPrepDown.setOnClickListener (new OnClickListener()
{
   public void onClick (View v) {
      decrementPrepTime();
   }
});
```

Mutant

Figure 4: OnClick Event Replacement mutant example

3.4. XML Mutation Operators

Android uses many XML files, not just the manifest file. XML files are used to define user interfaces, to store configuration data such as permissions, to define the default launch activity, and more. These three operators are unusual in that they do not modify executable code, but static XML.

*3.4.1. Activity Permission Deletion (**APD**)*

The Android operating system grants each app a set of permissions, such as the ability to access cameras or load location data from GPS sensors. These permissions are requested from the user when an app is first installed, and stored in the app's manifest file (*AndroidManifest.xml*). Some apps aggressively request unnecessary, even irrelevant, permissions, and many users click "OK" without paying attention to the details of these requested permissions when installing an app. This can create security and privacy vulnerabilities in Android systems.

APD mutants delete an app's permissions from its AndroidManifest.xml file, one at a time. If this mutant cannot be killed by any tests, it means that the app asked for a permission it did not need. For example, in Figure 5, the original program requests four permissions: WRITE_SETTINGS, WAKE_LOCK, MODIFY_AUDIO_SETTINGS, and VIBRATE. APD deletes the VIBRATE permission in the example mutant. Then, the app is not allowed to use the device's vibrator. A test that kills this mutant must cause the app to attempt to access the vibrator of the Android system.

```
<manifest xmlns:android=“http://schemas.android.com/apk/res/android”
     ... ...
  <uses-permission android:name=“android.permission.WRITE_SETTINGS”/>
  <uses-permission android:name=“android.permission.WAKE_LOCK” />
  <uses-permission android:name=“android.permission.MODIFY_AUDIO_SETTINGS” />
  <uses-permission android:name=“android.permission.VIBRATE”>
  </uses-permission>
</manifest>
```

Original

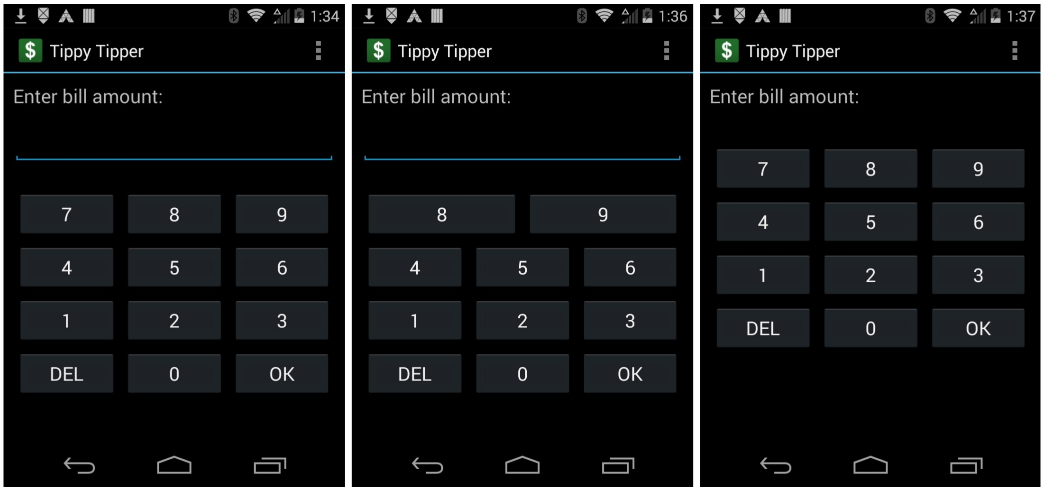
```
<manifest xmlns:android=“http://schemas.android.com/apk/res/android”
     ... ...
  <uses-permission android:name=“android.permission.WRITE_SETTINGS”/>
  <uses-permission android:name=“android.permission.WAKE_LOCK” />
  <uses-permission android:name=“android.permission.MODIFY_AUDIO_SETTINGS” />
<!- -
   <uses-permission android:name=“android.permission.VIBRATE”>
- ->
  </uses-permission>
</manifest>
```

Mutant

Figure 5: Activity Permission Deletion (APD) mutant example

*3.4.2. Button Widget Deletion (**BWD**)*

The button widget is used by nearly all Android apps in many ways. BWD deletes one button at a time from the XML layout file of the UI. Killing the BWD mutants requires tests that ensure that every button is successfully displayed. Figure 6 shows an original screen on the left, and two mutants on the right. The middle screen is a BWD mutant where the button



Original **BWD Mutant** **TWD Mutant**

Figure 6: Button Widget Deletion (BWD) and EditText Widget Deletion (TWD) mutant examples

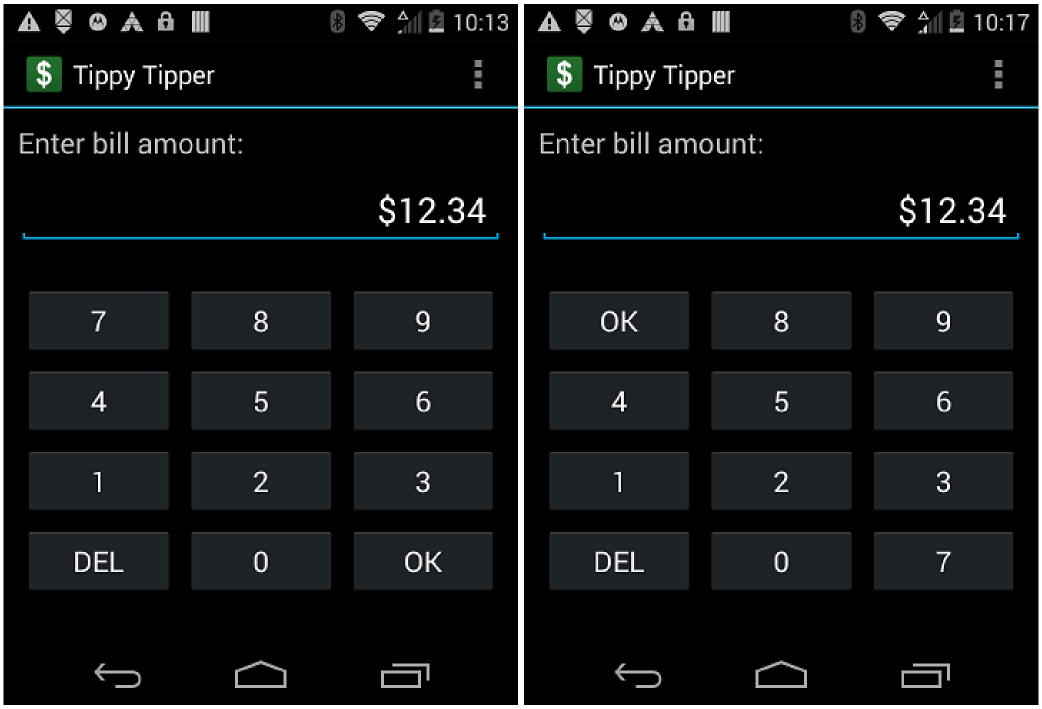
"7" is deleted from the UI. This mutation operator forces the tester to design tests that use each button in a way that affects the output behavior.

*3.4.3. EditText Widget Deletion (**TWD**)*

The EditText widget is used to display text to users. The TWD mutation operator removes each EditText widget, one at a time. The rightmost screen in Figure 6 shows an example TWD mutant where the bill amount cannot be displayed. To kill this mutant, a test must use the bill amount.

*3.4.4. Button Widget Switch (**BWS**)*

It is common for testers to design test cases to ensure an app works as expected with respect to its functional requirements, and evaluate the GUI structure as a secondary issue. However, Android apps are event-based, which means it is essential to display the GUI structure appropriately, as well as handling user events. Unlike BWD, BWS does not remove a button widget, but switches the locations of two buttons on the same screen. In this way, the function of a button is unaffected, but the GUI layout looks different from the original version. BWS requires the tester to design tests that deliberately check the location (either relative or absolute) of a button widget. Figure 7 illustrates an example of BWS mutant. The mutant on the right side switches the locations of button "7" and "OK."



Original **Mutant**

Figure 7: Button Widget Switch mutant example

BWS mutants require tests to check the location of a widget. Testers need to design tests that load the location of a widget, and compare it with an expected value, or other widgets' location, to ensure its correct location. For example, the code snippet in Figure 8 loads and compares the locations of two button widgets to ensure the OK button is displayed on the left of the Cancel button.

3.5. Mutation Operators Based on Common Faults

We started our efforts to design mutation operators by investigating bug reports and code change history logs on GitHub repositories. With the analysis on the repositories of open source Android apps, including DAVdroid [33], CosyDVR [34], URL evaluator for Android [35], and oandbackup [36], we observed several types of faults that were common across different apps. To cover these, we designed and implemented novel fault-based operators.

```
Button okButton = (Button) solo.getView (R.id.ok);
Button cancelButton = (Button) solo.getView (R.id.cancel);
int [ ] locationOfOK = new int [2];
int [ ] locationOfCancel = new int [2];
okButton.getLocationInWindow (locationOfOK);
cancelButton.getLocationInWindow (locationOfCancel);
assertTrue (“OK button is on the left of Cancel”, locationOfOK [0] < locationOfCancel [0]);
```

Figure 8: Test code to kill a Button Widget Switch mutant

*3.5.1. Fail on Null (**FON**)*

According to Arlt et al. [37], *NullPointerException* is one of the most common exceptions thrown in programs. A common cause is that developers forget to check if an object is null before accessing it. In our initial study on GitHub repositories, we found 80 corrections to one app, of which 52 were patching null-checking statements. FON mutants add a “fail on null” statement before each object is referenced. For String objects, FON also adds a “fail on empty” statement before objects are accessed. Figure 9 shows an example of an FON mutant. The mutated statement is inserted before accessing *members*. FON mutants are used to encourage the tester to design tests that make *members* null and trigger the “fail on null” statement.

```
List<ResourceType> res = new LinkedList<> ();
List<Member> members = collection.getMembers ();

for (WebDavResource member : members)
   res.add (newResource (member.getName (), member.getETag ()));
return res.toArray (new Resource[0]);
```

Original

```
List<ResourceType> res = new LinkedList<> ();
List<Member> members = collection.getMembers ();
failOnNull (members);
for (WebDavResource member : members)
   res.add (newResource (member.getName (), member.getETag ()));
return res.toArray (new Resource[0]);
```

Mutant

Figure 9: Fail on Null mutant example

*3.5.2. Orientation Lock (**ORL**)*

Mobile devices such as smartphones and tablets have the unique feature of being able to change the screen orientation. Thus, many apps change

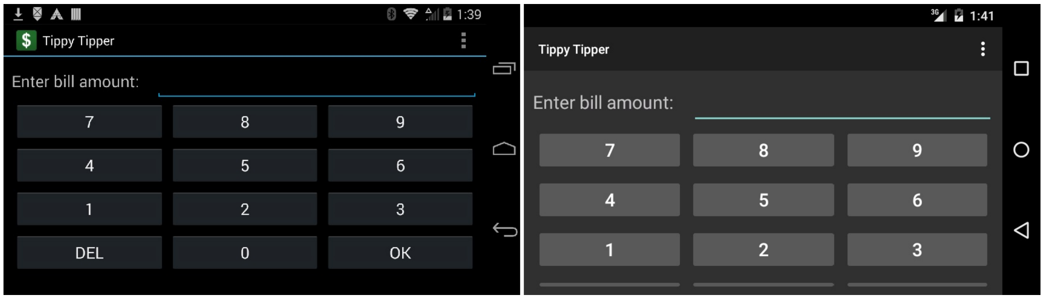


Figure 10: Fault in landscape orientation

the layout of the GUI when the orientation changes. For example, YouTube automatically switches to play video in full screen when the orientation is changed from portrait to landscape. However, Android devices are manufactured by different factories with various hardware specifications, using different screen sizes and resolutions. This makes switching the orientation difficult for the developers, in turn leading to many faults in Android apps.

Figure 10 shows a correct and a faulty version of TippyTipper with different orientations. Even though both devices properly display the GUI in portrait orientation, when switching to landscape orientation, as shown in Figure 10, the user is not able to see or click the button at the bottom or scroll down the screen.

ORL mutants freeze the orientation of an activity to be in portrait or landscape, by inserting a special *locking* statement into the source code. Only test cases that explicitly changes the orientation and checks whether the GUI structure is displayed as expected in both orientations can kill these mutants.

3.6. Mutation Operator Summary

These eleven mutation operators are defined on several unique and novel aspects of Android apps. These mutation operators cover many of Android app's novel features and allow a feasibility study.

Previous research efforts that defined new mutation operators have found that an initial set of operators can often be improved [16, 24, 38, 39, 40]. Improvements include adding additional operators that improve the fault detection ability of the resulting tests, eliminating redundant operators, and modifying operators to generate more, fewer, or better mutants. This will require experimental evaluations to identify mutation operators that do not lead to useful tests or that are redundant, as well as to identify additional useful operators.

4. Mutating Android Applications

Mutation analysis cannot be performed the same way for Android apps as for traditional Java programs. First, whereas Java mutation analysis tools mutate only Java files, Android operators also mutate XML layout and configuration files. Second, Android apps require additional processing before being deployed. Java mutation engines usually either mutate the source, then compile to bytecode class files, or compile to bytecode, then mutate the bytecode. The Java bytecode files are then dynamically linked by the language system during execution. Android apps have the additional requirement that each Android mutant must be compiled as an Android application package (APK) file so that it can be installed and executed on mobile devices and emulators. This significantly impacts the design of mutation analysis tools.

Figure 11 illustrates how our mutation analysis engine works. Below are the steps for conducting mutation analysis on Android apps. Note that steps 3, 4, 6, and 7 are different from traditional mutation testing processes.

1. First, the tester selects which mutation operators should be used. In addition to the eleven new Android operators defined in Section 3, we reuse 15 method level operators from muJava [25], and four deletion operators [41, 42]. The Android mutation analysis tool uses part of the muJava [25] mutant generation engine to implement these mutation operators[1].
2. For the operators that mutate Java code (both our new Android and the traditional muJava operators), the system modifies the original Java source code, and compiles them to bytecode class files.
3. XML mutation operators are applied directly to the XML file, creating a new copy of the file for each mutant. They are swapped into place for dynamic binding when the APK file is created.
4. For each mutated Java bytecode class file and XML file, the mutation system generates a mutated APK file by including the mutated source and other project files. Some mutants might cause compilation errors

---

[1] muJava is now open source on GitHub (*https://github.com/jeffoffutt/muJava*).

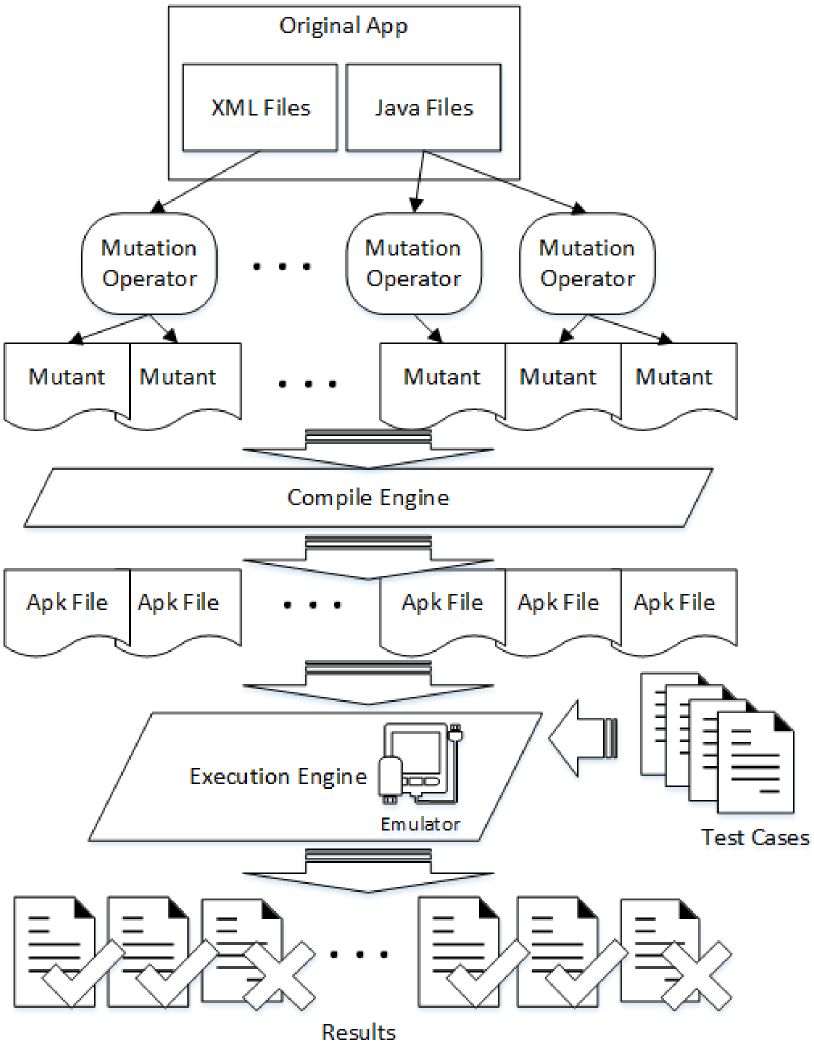


Figure 11: Performing mutation analysis on Android apps

(stillborn), which are discarded immediately and not used in the final results.

5. The Android testing framework extends JUnit [43] to support the testing of different types of Android components [44]. In addition, testers can write test cases with the support of external Android test automation frameworks, such as Robotium [45]. Our Android mutation analysis tool is implemented to run both kinds of test cases above. Tests

are either designed by the tester to target mutants, or an externally created set of pre-existing tests can be used. Each test is imported and compiled as an APK test file.

6. After generating mutants and compiling them to APK files, the system loads the original (non-mutated) version of the app under test into an emulator or onto a mobile device. Then the system executes all test cases on the original app and records the outputs as *expected* results. The results of the mutant executions are compared with the results of the original app to determine which mutants are killed.
7. Then, each mutant is loaded into an emulator or onto a mobile device. The mutation system executes all the test cases against the mutants and stores the outputs as the *actual results*. With the current tool, running Robotium test cases is very time-consuming. According to the Robotium developer, higher test execution speed may make the execution unstable on emulators [46]. In the emulator, each test requires hours to run against all mutants. In the future, we plan several optimizations to reduce this cost.
8. After collecting all the results, the mutation system compares the expected results with the actual results. If the actual result on a test differs from the expected result on the same test, that mutant is marked as having been killed by that test.
9. Finally, the mutation score is computed as a percentage of the mutants killed by the tests. Currently, the tool does not implement any heuristics to help identify equivalent mutants, so these must all be evaluated manually. Encouragingly, based on the evidence in Section 5, the Android mutation operators do not seem to create many equivalent mutants.

5. Empirical Evaluation

To evaluate our proposed approach, we developed a new mutation analysis tool that implements the mutation operators defined in Section 3. The tool was then used to generate mutants, compile the APK files, install them into emulators and Android devices, execute tests against the mutants, and compute and report the final results.

Based on other uses of mutation testing, we expect mutation of Android apps to be stronger than statement coverage, thus we use statement coverage as a comparison. Our empirical evaluation includes five phases: selecting empirical subjects, designing test data with 100% statement coverage, generating mutants with muJava and Android mutation operators, executing tests against mutants, and analyzing results.

Most testing is done using Android emulators, and most of our research has followed that example. However, to extend our previous work, and to check whether results are consistent between the emulator and hardware devices, we also used two Motorola MOTO G Android smartphones, one with the Dalvik Virtual Machine, and the other with ART. We executed tests in developer mode.

5.1. Empirical Subjects

Eight Android apps were selected as empirical subjects. These eight apps were used in previous papers [4, 8]. *TippyTipper* [47] is an Android app that can calculate tips after taxes are added and split bills among several customers. According to the Google Play store, the latest version 2.0 was released in December 2013 and currently has a 4.6 star rating from 761 users. We tested it by downloading the source from its homepage. *TippyTipper* has five Activities: TippyTipper, SplitBill, Total, Settings, and About. It also has one Service: TipCalculatorService. Figure 12 illustrates three Activities: TippyTipper is on the left, SplitBill is in the middle, and Total is on the right.

PasswordMaker Pro for Android [48] produces passwords for websites and other apps. It accepts a "master password" from the user, combines the URL or the name of the website requiring the password, and computes a unique password with hash algorithms. It has 23 classes in three different packages. On the Google Play store, the latest update was in January 2015, and it has a 3.7 star rating from 64 users.

MunchLife [49] is a counter application for tracking levels achieved while playing the card game Munchkin. Its latest version is 1.4.4, released in February 2014, with a 4.3 star rating from 242 users. *JustSit* [50] is a timer app with an alarm used for meditation. Its latest version is 0.3.3, released in July 2010, with a 3.8 star rating from 145 users. *Tipster* [51] is an app similar to TippyTipper that is used for splitting payment and calculating tips. It is an example from Darwin's book [52].

K-9 Mail [53] is an email client app with a rich set of useful features that are not offered by similar email clients. On the Google Play store, it has a 4.3 star rating from more than 155,000 reviewers, and several million user installations. Unlike our other apps, which were developed by a small number of programmers, K-9 Mail is developed and released by an open source community with hundreds of contributors.

Alarm Klock [54] is an alarm clock app with advanced and customizable features. It has a 4.5 star rating by 6291 reviews, and the Google Play store shows that the number of user installations is between 500,000 and 1,000,000.

Jamendo for Android [55] is an app for searching, streaming, and downloading free online music. We obtained it from F-Droid [56], a repository of free and open source Android apps. It is not currently available on the Google Play store, so we do not have review or download data.

We used twelve classes along with their corresponding XML layout files, and the AndroidManifest.xml files from the eight apps. TippyTipper, MunchLifeActivity, JustSit, PasswordMakerPro, TipsterActivity, ActivityAlarmClock, and HomeActivity are the main Activity classes of their apps. Other classes were chosen based on their features in the corresponding apps. For example, in the *TippyTipper* app, we chose the Activities SplitBill and Total because they provide features including splitting and calculating tips and taxes, and generate a rich set of mutants. We did not use the Activity About because it only displays information about the app without any additional functions, so could not create mutants. In addition, ColorPickerDialog of *K-9 Mail* is the only class that included event handlers for an OnTouch event. We used it to ensure the ETR operator was used.

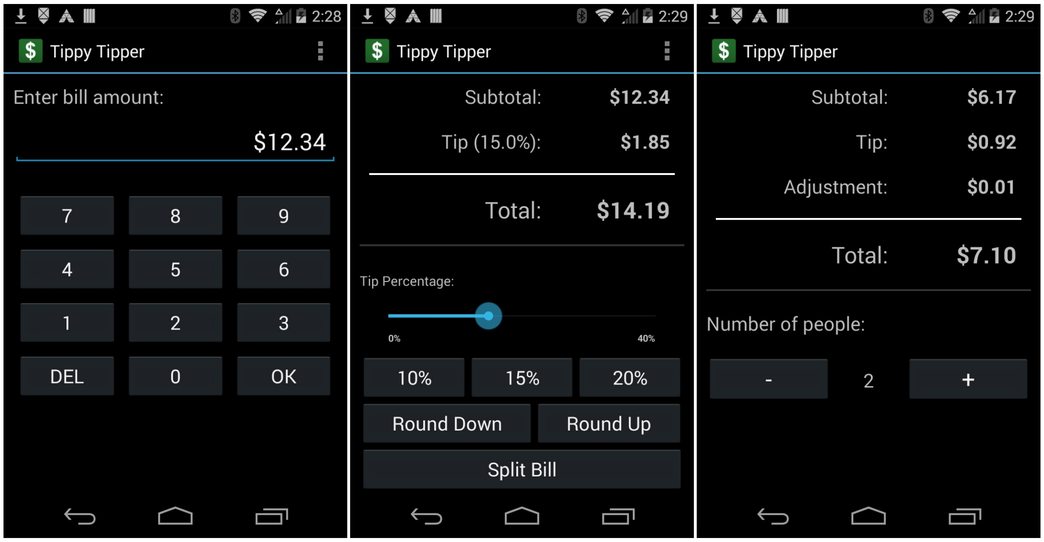
Details about the empirical subjects are in Table 2. The source lines of code (SLOC) and executable lines of code (ELOC) for the Android classes were calculated by Emma [57], and the LOCs for XML files were counted within the Android IDE. We also used the XML Document Object Model (DOM) parser to count the number of XML elements. We believe the number of elements is a better way to measure size of XML files than the number of lines.

The largest Java class is the main Activity of *PasswordMaker Pro*, PasswordMakerPro, with 606 SLOC. The smallest is the setting Activity of *MunchLife*, SettingsActivity, with 17 ELOC. The largest XML file is the AndroidManifest.xml of *K-9 Mail*, with 214 SLOC and 124 nodes.

Table 2 also lists the number of lines of dead code manually identified for each class. Our subjects had three types of dead code. First, if the default case is included in a switch-case block, but can never be reached with any user input, it is dead code. Second, if an event listener is designed for handling menu clicks, but no menu is on the screen, the entire listener class is dead code. Third, in a try-catch block, if it is impossible to throw and

App	**File**	**SLOC**	**ELOC**	**Lines of Dead Code**	**XML Elements**
TippyTipper	TippyTipper	239	103	1	
	main.xml	93	93		20
	SplitBill	134	63	6	
	SplitBill.xml	93	93		31
	Total (*a Java class*)	279	133	2	
	Total.xml	139	139		44
	AndroidManifest.xml	32	32		16
MunchLife	MunchLifeActivity	384	144	10	
	main.xml	58	58		12
	SettingsActivity	68	17	0	
	preferences.xml	25	25		5
	AndroidManifest.xml	32	32		10
JustSit	JustSit	444	207	30	
	main.xml	99	99		13
	JsSettings	61	22	0	
	JsSettings.xml	52	52		6
	AndroidManifest.xml	23	23		14
PasswordMaker Pro	PasswordMakerPro	606	343	26	
	main.xml	141	141		19
	AndroidManifest.xml	26	26		13
Tipster	TipsterActivity	297	115	0	
	main.xml	177	177		30
	AndroidManifest.xml	23	23		7
K-9 Mail	ColorPickerDialog	199	93	0	
	colorpicker_dialog.xml	59	59		7
	AndroidManifest.xml	214	214		124
Alarm Klock	ActivityAlarmClock	290	127	3	
	alarm_list.xml	39	39		6
	AndroidManifest.xml	53	53		35
Jamendo	HomeActivity	441	132	10	
	main.xml	66	66		10
	AndroidManifest.xml	146	146		93
	Total	**5032**	**3089**	**88**	**515**

Table 2: Details of empirical subjects



Enter bill amount: $12.34

Subtotal: $12.34
Tip (15.0%): $1.85
Total: $14.19
Tip Percentage:

Subtotal: $6.17
Tip: $0.92
Adjustment: $0.01
Total: $7.10
Number of people: 2

Figure 12: Three activities for TippyTipper

catch a required exception the entire catch block will be dead code.

5.2. Test Data Generation

We used pre-existing tests from our previous paper [9], and created new tests by hand. In our previous paper, EvoDroid [4], an evolutionary algorithm-based tool, generated 744 test cases for the main Activity of *TippyTipper* through multiple generations. We chose ten tests from the last generation, which covered 82% of the methods, 90% of the blocks, and 85% of the statements in the main Activity class, TippyTipper. We then added one additional test by hand to achieve full statement coverage.

For the other nine Android classes and their associated XML layout files, we manually designed test inputs to achieve 100% statement coverage (Table 2), excluding the dead code. All available test sets designed for each app were executed against APD mutants of AndroidManifest.xml files. For example, the test set for AndroidManifest.xml of *TippyTipper* consists of all the test cases designed to test the Activities of TippyTipper, SplitBill, and Total.

Because mobile devices and emulators usually have relatively fewer computation resources (e.g., less memory and lower CPU speed), sending test inputs directly to them without waiting for their responses to each user action is very likely to get inaccurate testing results. For example, if an action of clicking a button is sent before the button is completely rendered on the screen, the test will fail due to the failure of finding the button. Thus, to get accurate empirical results, we added code to our tests to wait for two seconds after each user action and before executing assertion statements.

5.3. Mutant Generation

App	**File**	**muJava Mutants**	**Android Mutants**
TippyTipper	TippyTipper	105	195
	SplitBill	124	37
	Total	231	104
	AndroidManifest.xml	n/a	4
MunchLife	MunchLifeActivity	534	151
	SettingsActivity	47	7
	AndroidManifest.xml	n/a	1
JustSit	JustSit	415	241
	JsSettings	28	29
PasswordMakerPro	PasswordMakerPro	515	379
Tipster	TipsterActivity	327	118
K-9 Mail	ColorPickerDialog	551	60
	AndroidManifest.xml	n/a	17
Alarm Klock	ActivityAlarmClock	161	235
	AndroidManifest.xml	n/a	6
Jamendo	HomeActivity	237	115
	AndroidManifest.xml	n/a	7
Total		3275	1706

Table 3: Mutants generated

According to the design of applying mutation analysis in Android apps in Section 4, we used 19 method-level mutation operators borrowed from muJava [25], and eleven Android operators designed in our research to generate mutants, and compile them into installable APK files. Generating a mutant and compiling it as an APK file took up to two seconds on a MacBook Pro with a 2.6 GHz Intel i7 processor and 16 GB memory.

Table 3 lists the results of mutants generation. Our system generated a total of 3275 mutants from the 19 method-level operators. The number of muJava mutants ranged from 28 (in JsSettings of *JustSit*) to 551 (in ColorPickerDialog of *K-9 Mail*). The eleven new Android mutation operators generated 1706 valid Android mutants for twelve Android classes along with their corresponding XML layout files, and five AndroidManifest.xml files (*TippyTipper*, *MunchLife*, *K-9 Mail*, *Alarm Klock*, and *Jamendo*). The number of Android mutants ranged from seven (in SettingsActivity of *MunchLife*) to 379 (in PasswordMakerPro of *PasswordMakerPro*), excluding AndroidManifest.xml files.

As stated in Section 2, a mutant that cannot be compiled into an APK file is called *stillborn*, and is not counted in the results. For example, the Activity class TippyTipper has 110 stillborn mutants in addition to 105 muJava and 195 Android mutants, for a total of 300 mutants. The entire TippyTipper app has 195+37+104+4 = 340 Android mutants and 105+124+231 = 460 muJava mutants (muJava does not generate any mutants for XML files). The 110 stillborn mutants are comprised of 36 AOIS mutants, 2 LOI mutants, 6 ITR mutants, and 66 ECR mutants. Some mutants are stillborn because of incorrect syntax. Other mutants are stillborn because Android apps use integers to identify pre-defined resources and values that are saved in a separate file. Some mutation operators mutate the identification integers, making it impossible for Android to locate these pre-defined values. In turn, this prevents APK files from being compiled.

Each Android app has an AndroidManifest.xml file, but three AndroidManifest.xml files (in subjects *JustSit*, *PasswordMakerPro*, and *Tipster*) did not have any mutants. Thus, they are not listed in Table 3.

5.4. Empirical results

We used our mutation analysis tool to load and execute 100% statement coverage test sets against all mutants. Table 4 summarizes results from running both muJava and Android mutants. Across all subjects, 1778 of 3275 muJava mutants and 530 of 1706 Android mutants were killed by the statement coverage test sets. Equivalent mutants were identified by hand analysis. The MS columns in Table 4 show mutation scores *after* equivalent mutants are filtered out. In other words, the percentages show how many mutants are killed relative to how many can be killed. The mutation scores for the muJava mutants ranged from 0.419 (in JustSit of *JustSit*) to 0.78 (in TipsterActivity of *Tipster*), with a mean of 0.622 and a median of 0.644. For Android mutants, the mutation scores ranged from 0.455 (in SplitBill of *TippyTipper*) to 0.885 (in HomeActivity of *Jamendo for Android*), with a mean of 0.666 and a median of 0.674, excluding the three AndroidManifest.xml files.

Table 5 shows results for each mutation operator. The first group contains results from the muJava traditional mutants. Arithmetic Operator Replacement (AORS) and Logical Operator Replacement (LOR) mutants have the

App	**File**	**muJava Mutants**				**Android Mutants**			
		Total	**Killed**	**Equiv.**	**MS**	**Total**	**Killed**	**Equiv.**	**MS**
TippyTipper	TippyTipper	105	71	4	0.703	195	85	41	0.552
	SplitBill	124	52	14	0.473	37	5	26	0.455
	Total	231	123	29	0.609	104	24	57	0.511
	AndroidManifest.xml	n/a				4	0	4	1.000
MunchLife	MunchLifeActivity	534	324	72	0.701	151	31	105	0.674
	SettingsActivity	47	19	8	0.487	7	2	3	0.500
	AndroidManifest.xml	n/a				1	1	0	1.000
JustSit	JustSit	415	153	50	0.419	241	59	174	0.881
	JsSettings	28	17	3	0.680	29	6	18	0.546
PasswordMakerPro	PasswordMakerPro	515	229	89	0.538	379	78	290	0.876
Tipster	TipsterActivity	327	234	27	0.780	118	22	88	0.733
K-9 Mail	ColorPickerDialog	551	271	56	0.547	60	13	45	0.867
	AndroidManifest.xml	n/a				17	3	0	0.176
Alarm Klock	ActivityAlarmClock	161	114	12	0.765	235	141	49	0.758
	AndroidManifest.xml	n/a				6	2	0	0.333
Jamendo	HomeActivity	237	171	13	0.763	115	54	54	0.885
	AndroidManifest.xml	n/a				7	4	0	0.571
Total		3275	1778	377	0.614	1706	530	954	0.705
Median		234	138	28	0.644	60	13	41	0.674
Mean		272.9	148.2	31.4	0.622	100.4	31.2	56.1	0.666

Table 4: Empirical results

lowest mutation scores of 0, meaning that none were killed by the statement coverage test sets. These two operators only generated five mutants, so this low percentage probably isn't meaningful. Among the Android mutation operators, none of the Button Widget Switch (BWS) mutants were killed. The highest mutation score among the traditional muJava mutants were for Conditional Operator Deletion (COD), 0.857. All of the Android mutants for OnTouch Event Replacement (ETR), Intent Payload Replacement (IPR), and Button Widget Deletion (BWD) were killed.

To assess whether the emulator had any effect on our tests, we ran the tests on different smartphones using Dalvik and ART. The mutation scores were identical in all three environments. However, the emulator is much slower than real devices, even with the Intel Hardware Accelerated Execution Manager (HAXM) installed.

5.5. Discussion

The APD operator (permission deletion) only applies to AndroidManifest.xml files. The *principle of least privilege* [58] requires that an app should only request necessary permissions from the Android system. If an app still works correctly after APD deletes its permissions (that is, the mutant is

Operator	**Killed Mutants**	**Equivalent Mutants**	**Live Mutants**	**Total Mutants**	**Mutation Scores**
Traditional Mutants					
AODU	3	0	1	4	0.750
AOIS	249	151	120	520	0.675
AOIU	253	17	112	382	0.693
AORB	113	4	71	188	0.614
AORS	0	0	1	1	0.000
CDL	22	9	18	49	0.550
COD	6	0	1	7	0.857
COI	70	4	55	129	0.560
COR	18	0	14	32	0.563
LOI	296	7	118	421	0.715
LOR	0	0	4	4	0.000
ODL	82	22	84	188	0.494
ROR	169	51	186	406	0.476
SDL	471	109	309	889	0.604
VDL	26	3	26	55	0.500
Subtotal	1778	377	1120	3275	0.614
Android Mutants					
APD	10	4	21	35	0.323
IPR	7	0	0	7	1.000
ITR	181	0	29	210	0.862
ECR	111	0	4	115	0.965
ETR	2	0	0	2	1.000
FON	146	949	25	1120	0.854
MDL	18	1	5	24	0.783
BWD	36	0	0	36	1.000
TWD	6	0	4	10	0.600
ORL	13	0	35	48	0.271
BWS	0	0	99	99	0.000
Subtotal	530	954	222	1706	0.705
Total	**2308**	**1331**	**1342**	**4981**	**0.632**

Table 5: Empirical results for each mutation operator

equivalent), the permission was unnecessary and granting it could create a security or privacy threat.

In our empirical study, none of the four APD mutants of *TippyTipper* were killed. We designed tests to cover only three out of five Activities in the app, thus testing could not show whether those Activities needed the permissions. To verify whether the permissions were needed, we conducted a detailed hand analysis of the needs of all the Activities, finding that none of the Activities used any of the four permissions requested (WRITE_SETTINGS, WAKE_LOCK, MODIFY_AUDIO_SETTINGS, and VIBRATE), leading us to conclude that *TippyTipper* does not need any of them. Thus we judged them to be equivalent. Additionally, fourteen live APD mutants of *K-9 Mail* were judged not equivalent after manual analysis.

In Table 5, 949 of 1120 (84.7%) FON mutants are equivalent, which is the highest in all mutation operators. This is because many objects in an app can never be null or empty. Thus, the "fail on null" statement is impossible to trigger. Our tool cannot decide if an object can be null when generating mutants. However, identifying and filtering these equivalent mutants by hand is straightforward and not time-consuming.

All the 99 BWS mutants were still alive after testing. as the statement coverage test sets could not ensure the locations (either relative, or absolute) of any button widgets.

In our previous paper [9], once the test set was augmented to achieve 100% statement coverage, the mutation score on the Android mutants was very high. Since mutation is usually much stronger than statement coverage, we interpreted this to mean that the initial Android mutants were not strong enough. The new operators used in this paper appear to have made this testing much stronger. 222 Android mutants were not killed, with an overall mutation score of 0.705. Additionally, the 100% statement coverage test sets were only able to kill 61% of muJava mutants, and 71% of Android mutants. This is more in line with previous mutation systems.

5.6. Threats to Validity

Our empirical evaluation has several threats to validity. First, dead code and equivalent mutants were identified manually by one person. Second, our implementation of the eleven proposed Android operators and Android mutation tool may include faults. To ensure they work as expected, we tested our tool constantly, and checked mutants generated by hand very carefully. Third, like most software engineering experiments, it is not possible to guarantee the representativeness of selected subjects. We tried to choose apps with different sizes, from different sources, and used in various domains. The fact that all the subjects were used by previous researchers provide consistency across multiple studies.

6. Related Work

This section describes relevant research in three areas: Android testing, mutation testing, and testing GUIs with mutation.

6.1. Android Testing

Android's development environment includes its own test framework [44], which extends the ubiquitous JUnit. Additionally, several testing automation frameworks are available to testers. Many testers use Robotium [45] for unit testing, system testing, and user acceptance testing. It is also compatible with other code coverage measurement tools, such as Emma and Cobertura. Thanks to its APIs that directly interact with Android GUI components by run-time binding, people with little knowledge of the implementation details can also write tests with Robotium. It is possible to test an app with Robotium even if only its APK file is available. However, to maintain a stable test execution on emulators and mobile devices, Robotium is set to run tests at a relatively low speed. All test sets used in our empirical study are designed with Robotium. Another framework for Android apps is Robolectric [59], which runs on the Java VM, instead of Dalvik or ART. It splits tests from the emulator, making it possible to run tests by directly referencing the Android library files. In testing Android apps, one challenge is the variety of hardware specifications, e.g., different screen sizes and resolutions. Selendroid [60] enables testers to distribute their tests across multiple emulators with different configurations. All these frameworks automate execution, but none supports test value generation, test criteria, or any other type of test design.

Several research papers have been based on random test value creation. Amalfitano et al. [5, 6] presented an approach that starts with random inputs, then uses a code-crawling algorithm to generate test cases. Hu and Neamtiu [61] generated GUI test inputs randomly and executed them with Android Monkey. They also collected and categorized faults from ten open source Android apps, and categorized Android faults into eight types: *activity error*, *event error*, *dynamic type error*, *unhandled exceptions*, *API error*, *I/O error*, *concurrency error*, and *others*. These categories are too general for use in mutation analysis.

The tool Dynodroid [8] creates random values and sequences of events, and uses heuristics to increase the speed of Android Monkey.

Some researchers use model-based approaches to generate tests for Android apps. By employing Android Monkey, *TEMA* [62] uses state machines (labeled state transition systems) to generate test sequences. However, two levels of state machines (action machine and refinement machine) need to be created by hand. *MobiGUITAR* [29] automates GUI-driven testing of Android apps by extracting run-time states of GUI widgets, and generates tests with abstraction of models. Compared with Android Monkey and Dynodroid, *MobiGUITAR* was reported to detect more faults. *ORBIT* [30] creates a GUI model of the app and then generates tests. A^3E [7] uses static taint analysis algorithms to build a model of the app, which is then used to automatically explore the Activities in the app. These papers focus on constructing models from which tests can be designed, as opposed to applying a test criterion such as mutation.

Some papers explore and extend symbolic execution into testing Android apps. Mirzaei et al. [63] created stubs and mock classes to make Android apps run on Java PathFinder (JPF) [64]. Merwe et al. [65, 66] developed JPF-Android by extending JPF to verify Android apps, but the state explosion problem made it difficult to generate complex test inputs. Jensen et al. [67] combined symbolic execution with test sequence generation to support system testing. Their goal was to find valid sequences and inputs that would reach locations in the code. Our research tries to maximize test case effectiveness through mutation testing, an exceptionally strong coverage criterion. Anand et al. [68] used dynamic symbolic execution [69, 70] in the form of concolic testing [71] to test an Android library. Their testing used pixel coordinates to identify valid GUI events.

Finally, several papers applied evolutionary algorithms [4, 72] to test Android apps. They focused on generating inputs for GUI testing of Android apps, instead of using test criteria.

6.2. Mutation Testing

Mutation testing has been applied to many languages, including Fortran 77 [16, 21], C [73], Java [25, 74], Javascript [75], AspectJ [76], and web applications [39]. Several papers also extend mutation analysis to model-level, such as Finite State Machines [77, 78], statecharts [79], Petri nets [80], timed automata [81], and Aspect-oriented models [82]. For GUI-based applications, a specific set of mutation operators [83] are also proposed. However, to our knowledge, mutation testing has not previously been applied to mobile apps.

To test messages transmitted between web components, Lee and Offutt applied mutation testing to XML data by defining mutation operators to mutate the interaction recorded in XML files [84]. Test cases are designed to detect the changes made to XML messages. Offutt and Xu approached the problem of input data validation for web services by designing mutation operators that modified XML schemas [85]. The approach was verified through experiments on web service applications. The paper used the term *perturbation* instead of mutation to emphasize that the mutation operators were *perturbing* the input space. Our approach is slightly different. We mutate XML files, but the XML files we mutate do not define input data, they help configure the app.

Mutation testing subsumes other test criteria by incorporating appropriate mutation operators. Designing effective mutation operators is the most important task when applying mutation to new technology, because the operators directly determine the strength of the resulting tests.

The cost of mutation testing is very high, as it has the largest number of test requirements among all of test coverage criteria. To reduce this cost, three types of approaches are used: *do-fewer*, *do-smarter*, and *do-faster* [86, 87]. As a *do-fewer* approach, *selective mutation* was proposed by Wong and Mathur by only choosing a subset of mutation operators [88, 89]. The muJava tool selects 15 operators to preserve almost the same test coverage as non-selected mutation [25]. Additionally, empirical studies in both Java and C show that Statement Deletion mutation operator (SDL) is able to result in very effective tests with much cheaper cost [41, 42]. Deletion operators are also included in the empirical study of this paper.

6.3. Testing GUIs with Mutation

The first research paper to use mutation to test GUIs was by Oliveira et al. in 2015 [83]. The paper introduced a way to design comprehensive tests for GUI-related programs, and also defined several challenges with respect to how to killing mutants. In practice, testers usually design tests to check the presence of GUI widgets. For example, with the help of Robotium, the statement

assertTrue(solo.searchButton("OK"));

checks whether a button widget with text "OK" is displayed on the screen.

To kill deletion-related GUI mutants, such as BWD mutants, testers need to design test code similar to the one above to ensure all widgets are correctly displayed, regardless of their locations.

Furthermore, computer vision techniques and graphical test oracles have been used to test GUI and Android apps [90, 91, 92]. These techniques can also be used to kill GUI-related mutants, though they require more empirical studies to validate their value. We can imagine that these killing tests could be especially useful in regression testing, such as when features are added, removed, or adjusted.

7. Conclusions and Future Work

This paper proposes an innovative approach to test Android apps by using mutation analysis. We defined new mutation operators specific to Android apps, implemented them in a mutation analysis tool, and conducted an experiment with eight Android apps. The results show that mutation testing can be extended to accommodate program structures novel to Android development. Our approach provides more comprehensive testing for Android apps by considering not only Java characteristics, but also XML layout, configuration information, and other Android characteristics.

In comparison with our previous paper [9], this paper contributes three additional novel Android mutation operators (FON, BWS, and ORL). We also extended the empirical study from one Android app to eight. In addition to using an Android emulator, we ran our tests on two Android devices with different runtime systems (Dalvik and ART). The tests behaved identically on 100% statement coverage tests in all the three environments.

While promising, several research questions remain unanswered. An important evaluation, currently being planned, is to do a full fault study. We will generate tests to kill all non-equivalent mutants, then evaluate those tests to determine how many faults the tests detect, and compare with tests generated for other criteria (possibly statement or branch coverage).

As mentioned in Section 6, Hu and Neamtiu [61] categorized Android faults into eight types. However, these categories appear to be too general to be applied in mutation analysis. Instead, we defined new Android mutation operators based on the unique characteristics of Android applications.

A well defined Android fault model could improve the power of our mutation operators by providing a reference against which to evaluate mutation operators. We are currently developing an Android fault model by investigating actual faults in open source repositories.

This paper defined eleven Android mutation operators that mutate Java source code, XML layout files, and Android permissions. However, we have not yet considered all aspects of Android apps. For instance, one important distinct characteristic of mobile apps is that they are context-aware. Context-aware apps behave differently when the phone is moving in a vehicle and sitting at a desk. This difference in behavior is not reflected directly in the app code; rather the difference is in how often the app receives an event notification about location. In a sense, location event notifications are inputs that should be modeled as part of the test. We plan additional mutation operators to test context-aware behaviors.

We are still improving our Android mutation tool. In particular, we need to make our tool generate fewer stillborn mutants, fewer mutants that immediately crash, and more hard to kill mutants. Also, we hope to employ external well-established frameworks, such as Xposed [93], which has the potential to speed up mutation analysis.

The cost of mutation testing Android apps is especially expensive due to the slow speed of Android test execution with Robotium. A single iteration of an experiment required more than 20 hours. Performance could be improved by evaluating mutants in parallel, finding or building a faster test framework, or using fewer mutants. Work in general program mutation suggests that only a small number of generated mutants are necessary [40]; this result may extend to Android mutation as well. We are currently evaluating these approaches.

Acknowledgment

Offutt is partially funded by The Knowledge Foundation (KKS) through project 20130085, Testing of Critical System Characteristics (TOCSYC).

References

[1] Gartner, Gartner says sales of smartphones grew 20 percent in third quarter of 2014, Online, 2014. `https://www.gartner.com/newsroom/id/2944819/`, last access January 2015.

[2] Google Play, 2015. `https://play.google.com/store`, last access January 2015.

[3] P. Bhattacharya, L. Ulanova, I. Neamtiu, S. C. Koduru, An empirical analysis of bug reports and bug fixing in open source Android apps, in: 2013 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 133–143.

[4] R. Mahmood, N. Mirzaei, S. Malek, Evodroid: Segmented evolutionary testing of Android apps, in: Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '14, ACM, Hong Kong, China, 2014.

[5] D. Amalfitano, A. Fasolino, P. Tramontana, A GUI crawling-based technique for Android mobile application testing, in: Third International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software, pp. 252–261.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, A. M. Memon, Using GUI ripping for automated testing of Android applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, ACM, New York, NY, USA, 2012, pp. 258–261.

[7] T. Azim, I. Neamtiu, Targeted and depth-first exploration for systematic testing of Android apps, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 641–660.

[8] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: An input generation system for Android apps, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 224–234.

[9] L. Deng, N. Mirzaei, P. Ammann, J. Offutt, Towards mutation analysis of Android apps, in: Eleventh IEEE Workshop on Mutation Analysis (Mutation 2015), Graz, Austria, pp. 1–10.

[10] Dalvik - code and documentation from Android's VM team, 2014. `http://code.google.com/p/dalvik/`, last access January 2015.

[11] ART and Dalvik, 2014. `https://source.android.com/devices/tech/dalvik/index.html`, last access March 2015.

[12] Android developers guide, 2015. `http://developer.android.com/guide/topics/fundamentals.html`, last access January 2015.

[13] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer 11 (1978) 34–41.

[14] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, UK, 2008. ISBN 978-0-521-88038-1.

[15] L. J. Morell, A theory of fault-based testing, IEEE Transactions on Software Engineering 16 (1990) 844–857.

[16] R. A. DeMillo, J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (1991) 900–910.

[17] J. Offutt, J. Payne, J. M. Voas, Mutation Operators for Ada, Technical Report ISSE-TR-96-09, Department of Information and Software Engineering, George Mason University, Fairfax VA, 1996. Http://www.cs.gmu.edu/∼tr_admin/.

[18] G. M. Kapfhammer, P. McMinn, C. J. Wright, Search-based testing of relational schema integrity constraints across multiple database management systems, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 31–40.

[19] L. Bottaci, Type sensitive application of mutation operators for dynamically typed programs, in: 5th International Workshop on Mutation Analysis (Mutation 2010), pp. 126–131.

[20] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, G. Spafford, Design of Mutant Operators for the C Programming Language, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette IN, 1989.

[21] K. N. King, J. Offutt, A Fortran language system for mutation-based software testing, Software-Practice and Experience 21 (1991) 685–718.

[22] Y.-S. Ma, Y.-R. Kwon, J. Offutt, Inter-class mutation operators for Java, in: Proceedings of the 13th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Annapolis MD, 2002, pp. 352–363.

[23] S. Kim, J. A. Clark, J. A. McDermid, Investigating the applicability of traditional test adequacy criteria for object-oriented programs, in: Proceedings of ObjectDays 2000.

[24] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of muJava, in: Workshop on Automation of Software Test (AST 2006), Shanghai, China, pp. 78–84.

[25] Y.-S. Ma, J. Offutt, Y.-R. Kwon, MuJava : An automated class mutation system, Software Testing, Verification, and Reliability, Wiley 15 (2005) 97–133.

[26] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, C. Hutchinson, A fault model for subtype inheritance and polymorphism, in: Proceedings of the 12th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Hong Kong China, 2001, pp. 84–93.

[27] Android App Development Tutorial, 2015. `http://www.codelearn.org/android-tutorial/android-introduction`, last access November 2015.

[28] Activity Testing: What to Test, 2015. `http://developer.android.com/tools/testing/activity_testing.html#WhatToTest`, last access November 2015.

[29] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, A. Memon, Mobiguitar–A tool for automated model-based testing of mobile apps, IEEE Software 32 (2014) 53–59.

[30] W. Yang, M. R. Prasad, T. Xie, A grey-box approach for automated GUI-model generation of mobile applications, in: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 250–265.

[31] Android: What to Test, 2015. `http://developer.android.com/tools/testing/what_to_test.html`, last access November 2015.

[32] Android Intent, 2015. `http://developer.android.com/reference/android/content/Intent.html`, last access January 2015.

[33] DAVdroid, 2015. `https://github.com/bitfireAT/davdroid`, last access May 2015.

[34] URL evaluator for Android, 2015. `https://github.com/nicolassmith/urlevaluator`, last access May 2015.

[35] CosyDVR, 2015. `https://github.com/sergstetsuk/CosyDVR`, last access May 2015.

[36] oandbackup, 2015. `https://github.com/jensstein/oandbackup`, last access May 2015.

[37] S. Arlt, C. Rubio-Gonzalez, P. Rummer, M. Schaf, N. Shankar, The gradual verifier, in: J. Badger, K. Rozier (Eds.), NASA Formal Methods, volume 8430 of *Lecture Notes in Computer Science*, Springer International Publishing, 2014, pp. 313–327.

[38] J. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf, An experimental determination of sufficient mutation operators, ACM Transactions on Software Engineering Methodology 5 (1996) 99–118.

[39] U. Praphamontripong, J. Offutt, Applying mutation testing to web applications, in: Sixth Workshop on Mutation Analysis (IEEE Mutation 2010), Paris, France.

[40] P. Ammann, M. E. Delamaro, J. Offutt, Establishing theoretical minimal sets of mutants, in: 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014), Cleveland, Ohio.

[41] L. Deng, J. Offutt, N. Li, Empirical evaluation of the statement deletion mutation operator, in: 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg.

[42] M. E. Delamaro, J. Offutt, P. Ammann, Designing deletion mutation operators, in: 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014), Cleveland, Ohio.

[43] JUnit, 2014. `http://junit.org`, last access January 2015.

[44] Android testing framework, 2015. `http://developer.android.com/guide/topics/testing/`, last access January 2015.

[45] Robotium, 2015. `http://code.google.com/p/robotium/`, last access January 2015.

[46] How to increase speed of tests in Robotium?, 2012. `https://github.com/robotiumtech/robotium/issues/296`, last access December 2015.

[47] TippyTipper, 2013. `https://code.google.com/p/tippytipper`, last access January 2015.

[48] PasswordMakerProForAndroidActivity, 2015. `https://play.google.com/store/apps/details?id=org.passwordmaker.android`, last access July 2015.

[49] MunchLife, 2014. `https://play.google.com/store/apps/details?id=info.bpace.munchlife`, last access July 2015.

[50] JustSit, 2010. `https://play.google.com/store/apps/details?id=com.brocktice.JustSit`, last access July 2015.

[51] I. Darwin, Tipster, 2014. `https://github.com/IanDarwin/Android-Cookbook-Examples/tree/master/Tipster`, last access July 2015.

[52] I. Darwin, Android Cookbook, O'Reilly Media, 2012. ISBN 9978-1449388416.

[53] K-9 Mail, 2015. `https://play.google.com/store/apps/details?id=com.fsck.k9`, last access November 2015.

[54] Alarm Klock, 2015. `https://play.google.com/store/apps/details?id=com.angrydoughnuts.android.alarmclock`, last access November 2015.

[55] Jamendo for Android, 2015. `http://telecapoland.github.io/jamendo-android/`, last access November 2015.

[56] F-Droid, 2015. `https://f-droid.org`, last access November 2015.

[57] V. Roubtsov, Emma, Online, 2006. `http://emma.sourceforge.net/`, last access January 2015.

[58] J. Saltzer, M. Schroeder, The protection of information in computer systems, Proceedings of the IEEE 63 (1975) 1278–1308.

[59] Robolectric, 2015. `https://github.com/robolectric/robolectric`, last access January 2015.

[60] Selendroid, 2015. `http://selendroid.io`, last access July 2015.

[61] C. Hu, I. Neamtiu, Automating GUI testing for Android applications, in: Proceedings of the 6th International Workshop on Automation of Software Test, AST '11, ACM, New York, NY, USA, 2011, pp. 77–83.

[62] T. Takala, M. Katara, J. Harty, Experiences of system-level model-based GUI testing of an android application, in: 4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011), pp. 377–386.

[63] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, R. Mahmood, Testing Android apps through symbolic execution, SIGSOFT Software Engineering Notes 37 (2012) 1–5.

[64] Java PathFinder, 2007. `http://babelfish.arc.nasa.gov/trac/jpf/`, last access January 2015.

[65] H. van der Merwe, B. van der Merwe, W. Visser, Verifying android applications using java pathfinder, SIGSOFT Software Engineering Notes 37 (2012) 1–5.

[66] H. van der Merwe, B. van der Merwe, W. Visser, Execution and property specifications for JPF-Android, SIGSOFT Softw. Eng. Notes 39 (2014) 1–5.

[67] C. S. Jensen, M. R. Prasad, A. Møller, Automated testing with targeted event sequence generation, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, ACM, New York, NY, USA, 2013, pp. 67–77.

[68] S. Anand, M. Naik, M. J. Harrold, H. Yang, Automated concolic testing of smartphone apps, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 59:1–59:11.

[69] B. Korel, A dynamic approach of test data generation, in: Conference on Software Maintenance-1990, San Diego, CA, pp. 311–317.

[70] J. Offutt, Z. Jin, J. Pan, The dynamic domain reduction approach to test data generation, Software-Practice and Experience 29 (1999) 167–193.

[71] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, Chicago Illinois, USA, pp. 213–223.

[72] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, A. Stavrou, A whitebox approach for automated security testing of Android applications on the cloud, in: 2012 7th International Workshop on Automation of Software Test (AST), pp. 22–28.

[73] M. E. Delamaro, J. C. Maldonado, Proteum-A tool for the assessment of test adequacy for C programs, in: Proceedings of the Conference on Performability in Computing Systems (PCS 96), New Brunswick, NJ, pp. 79–95.

[74] S. Kim, J. A. Clark, J. A. McDermid, Investigating the effectiveness of object-oriented strategies with the mutation method, in: Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, pp. 4–100. Wiley's Software Testing, Verification, and Reliability, December 2001.

[75] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Efficient JavaScript mutation testing, in: 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), pp. 74–83.

[76] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, C. V. Lopes, Testing aspect-oriented programming pointcut descriptors, in: Proceedings of the 2nd workshop on testing aspect-oriented programs, ACM, pp. 33–38.

[77] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, P. C. Masiero, Mutation analysis testing for finite state machines, in: 5th IEEE International Symposium on Software Reliability Engineering (ISSRE 94), Monterey, CA, pp. 220–229.

[78] R. Hierons, M. Merayo, Mutation testing from probabilistic finite state machines, in: Third Workshop on Mutation Analysis (IEEE Mutation 2007), Windsor, UK, pp. 141–150.

[79] M. Trakhtenbrot, New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models, in: Third Workshop on Mutation Analysis (IEEE Mutation 2007), Windsor, UK, pp. 151–160.

[80] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, E. W. Wong, Mutation analysis applied to validate specifications based on Petri nets, in: Proceedings of the 8th International Conference on Formal Description Techniques (FORTE'95), Quebec, Canada, pp. 329–337.

[81] R. Nilsson, J. Offutt, J. Mellin, Test case generation for mutation-based testing of timeliness, in: Proceedings of the 2nd International Workshop on Model Based Testing, Vienna, Austria, pp. 102–121.

[82] B. Lindstrom, S. Andler, J. Offutt, P. Pettersson, D. Sundmark, Mutating aspect-oriented models to test cross-cutting concerns, in: Eleventh IEEE Workshop on Mutation Analysis (Mutation 2015).

[83] R. Oliveira, E. Alegroth, Z. Gao, A. Memon, Definition and evaluation of mutation operators for GUI-level mutation analysis, in: Eleventh IEEE Workshop on Mutation Analysis (Mutation 2015), pp. 1–10.

[84] S. C. Lee, J. Offutt, Generating test cases for XML-based web component interactions using mutation analysis, in: 2001 12th International Symposium on Software Reliability Engineering (ISSRE 2001), pp. 200–209.

[85] J. Offutt, W. Xu, Testing web services by XML perturbation, in: Proceedings of the 16th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Chicago, IL, 2005.

[86] R. Untch, Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method, Ph.D. thesis, Clemson University, Clemson SC, 1995. Clemson Department of Computer Science Technical report 95-115.

[87] J. Offutt, R. Untch, Mutation 2000: Uniting the orthogonal, in: Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, pp. 45–55.

[88] W. E. Wong, M. E. Delamaro, J. C. Maldonado, A. P. Mathur, Constrained mutation in C programs, in: Proceedings of the 8th Brazilian Symposium on Software Engineering, Curitiba, Brazil, pp. 439–452.

[89] W. E. Wong, A. P. Mathur, Reducing the cost of mutation testing: An empirical study, Journal of Systems and Software, Elsevier 31 (1995) 185–196.

[90] T.-H. Chang, T. Yeh, R. C. Miller, GUI testing using computer vision, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10, ACM, New York, NY, USA, 2010, pp. 1535–1544.

[91] M. E. Delamaro, F. de Lourdes dos Santos Nunes, R. A. P. de Oliveira, Using concepts of content-based image retrieval to implement graphical testing oracles, Software Testing, Verification and Reliability 23 (2013) 171–198.

[92] Y.-D. Lin, J. Rojas, E.-H. Chu, Y.-C. Lai, On the accuracy, efficiency, and reusability of automated test oracles for android devices, IEEE Transactions on Software Engineering 40 (2014) 957–970.

[93] Xposed Framework, 2015. `http://repo.xposed.info`, last access July 2015.