Towards Mutation Analysis of Android Apps

Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt
Department of Computer Science
George Mason University
Fairfax, Virginia, USA
{ldeng2, nmirzaei, pammann, offutt}@gmu.edu

***Abstract*—Android applications (*apps*) have the highest number of releases, purchases, and downloads among mobile apps. However, quality is a known problem, and hence there is significant research interest in better methods for testing Android apps. We identify three reasons to extend mutation testing to Android apps. First, current testing approaches for Android apps use simple coverage criteria such as statement coverage; extending mutation coverage to Android apps promises more sophisticated testing. Second, testing researchers inventing other test methods for Android apps need to evaluate the quality of their test selection strategies, which mutation excels at. Finally, some approaches to test generation for Android apps, specifically combinatorial testing approaches, generate very large numbers of tests. This is particularly problematic because running Android tests is slow. For these reasons, this paper proposes an innovative mutation analysis approach 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 behavior. We have implemented a prototype tool for generating, installing, and executing mutants on Android systems. We report preliminary results that show that mutation testing is feasible for Android apps, and we identify challenges that need to be addressed for mutation analysis to be effective.**

***Index Terms*—Android, Software Testing, Mutation Testing**

I. 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 tremendously as more platforms become available, 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%) [26]. Over a million apps are available to Android users on the Google Play store, the most widely used Android app store [6], 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. Although part of the problem is a lack of software engineering process (little or no testing), a significant technical problem also exists. Android apps involve several new programming features and we have very little knowledge for 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 [40]. Although several researchers have proposed improved test techniques [13], [14], [18], [38], [40], these have not reached actual practice.

The goal of this 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 II). Specifically, we propose to use mutation analysis, a high end testing technique that is known for yielding 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 ones to keep and which ones 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 eight novel mutation operators specific to Android apps.
- It evaluates these mutation operators on an example Android app.
- It identifies future research areas for mutation analysis of Android apps.

This paper is organized as follows. Section II describes how Android apps are programmed, including some of the unique aspects of programming in the framework, and introduces how mutation analysis works. Section III defines eight novel mutation operators that mutate new programming features such as the *Intent* and *event handlers*. Section IV outlines how mutation analysis is applied in the Android framework, which is quite different from traditional languages such as

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th International Workshop on Mutation Analysis (Mutation 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

Java. Section V presents an Android app, shows how mutation analysis can be used to test it, and provides preliminary results of the empirical study. The paper concludes with an overview of the related research in Section VI, and a discussion of our planned future work in Section VII.

II. Background

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

A. *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 the device. Android includes an operating system based on Linux, including middleware, pre-installed applications, and system libraries. Android used the Dalvik Virtual Machine [5] 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 [4]. 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 XML and provide information about the app to the ADF, including configuration 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 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 is used to describe 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 *Intent* messages. This is enabled by having

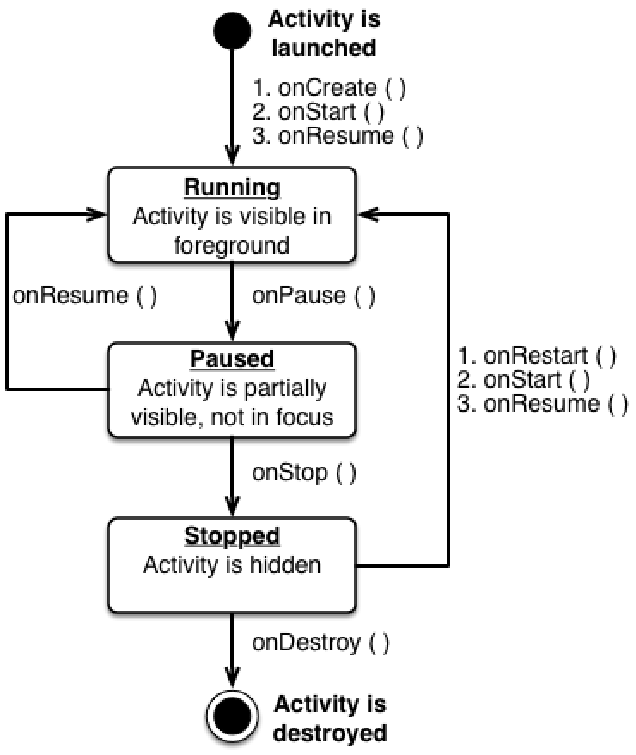

Fig. 1. Lifecycle of Activity in Android apps

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 [1]. 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()* and *onResume()* or it can exit with an *onDestroy()* event. ADF calls lifecycle event handlers and are integral to our research, as explained later.

B. *Mutation Analysis*

This paper proposes the use of mutation of Android app components to design effective tests. Mutation testing modifies a software artifact such as a program, requirements specification, or a configuration file to create new versions called *mutants* [21]. 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 different outputs on the original and each mutated version, called *killing* the mutant. For example, the *ROR* operator for Java 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 [16]. 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 the test suite to measure the percentage of mutants killed by the tests. This is called the *mutation adequacy score*. Mutation testing has been measured 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*) [16], [22], [43].

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 because the change makes the program syntactically incorrect. While these *stillborn* 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 [12], [31], [33], [36]. Mutation operators have also been defined for aspect-oriented software in AspectJ [35], and applied to modeling languages such as finite state machines [24], [28], statecharts [51], Petri nets [25], and timed automata [44]. Mutation operators for Android apps focus on the novel features of Android, including the manifest file, activities, services, etc.

III. 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 [37], [47] were based on a previous fault model by Alexander [45]. A second 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. Since we have not been found a fault model for Android apps, we used the second approach. Thus, we designed operators to fully cover the novel features used in Android apps. The following subsections define eight operators, divided into four categories: Intent related, event handler related, activity lifecycle related, and XML related.

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

TABLE I
IPR default values

A. Intent Mutation Operators

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

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 I. Objects with primitive types, such as int, short, long, etc., are replaced by the value of zero, String objects are replaced by empty strings, and boolean variables are replaced by both true and false. Figure 2 shows an example IPR mutant. The String object *message* is replaced with an empty String. This mutant challenges the tester to design test cases to ensure the value passed by an Intent object is correct.

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.

B. 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.

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.

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.

```
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

Fig. 2. Intent Payload Replacement Mutation Operator

```
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

Fig. 3. Intent Target Replacement 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

Fig. 4. Intent Target Replacement Mutation Operator

C. *An Activity Lifecycle Mutation Operator*

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

1) Lifecycle Method Deletion (MDL): Developers override these methods to define transitions among states. MDL deletes each overriding method to force Android to call the version in 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 [37], but only considers the methods related to the Activity lifecycle.

D. *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.

1) Button Widget Deletion (BWD): The button widget is used by nearly all Android apps in many ways. BWD deletes buttons one 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 5 shows an original screen on the left, and two mutants on the right. The middle screen is a BWD mutant where the button "7" is deleted from

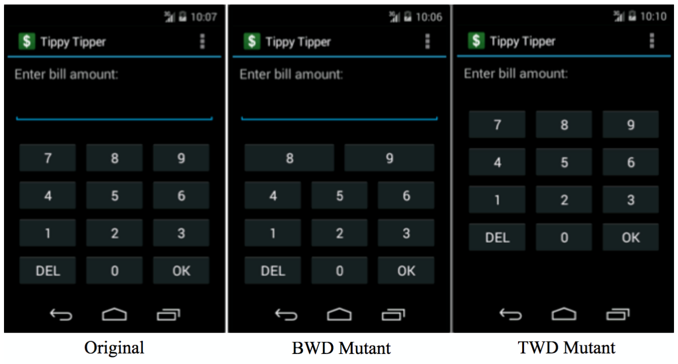

Fig. 5. Button Widget Deletion (BWD) and EditText Widget Deletion (TWD) Example

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

2) 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 5 shows an example TWD mutant where the bill amount cannot be displayed. To kill this mutant, a test must use the bill amount.

3) 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*).

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. This is a security vulnerability that can threaten the system beyond the app.

These eight mutation operators are defined on several unique and novel aspects of Android apps. Although a beginning, these should not, as yet, be considered a complete set of mutation operators.

IV. Approach Overview

Mutation analysis cannot be performed the same way for Android apps as for traditional Java programs. One reason the process must be different is that, whereas Java mutation analysis tools mutate only Java files, we have designed Android operators that also apply to XML layout and configuration files. A second reason is because Android apps require additional processing before being deployed. Traditional Java mutation analysis tools typically compile mutated Java source files to bytecode Java class files. 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 has a significant impact on how mutation analysis tools run.

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

1) First, the tester selects which mutation operators should be used. Our current mutation analysis engine implements the eight new Android operators proposed in Section III. Additionally, we reuse 15 types of method level muJava selective operators from muJava [37], and four deletion operators [20], [23]. The Android mutation analysis tool uses part of the muJava [37] mutant generation engine to implement these mutation operators.
2) For the traditional Java mutation 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. These stillborn mutants are discarded immediately and not used in the final results.
5) The Android testing framework extends JUnit [7] to support the testing of different types of Android components [3]. In addition, testers can write test cases with the support of external Android test automation frameworks, such as Robotium [9]. 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 developer of Robotium, higher test execution speed may make the execution unstable on emulators. In fact, each test requires hours to run against all mutants. We plan several optimizations to the tool to reduce this cost in

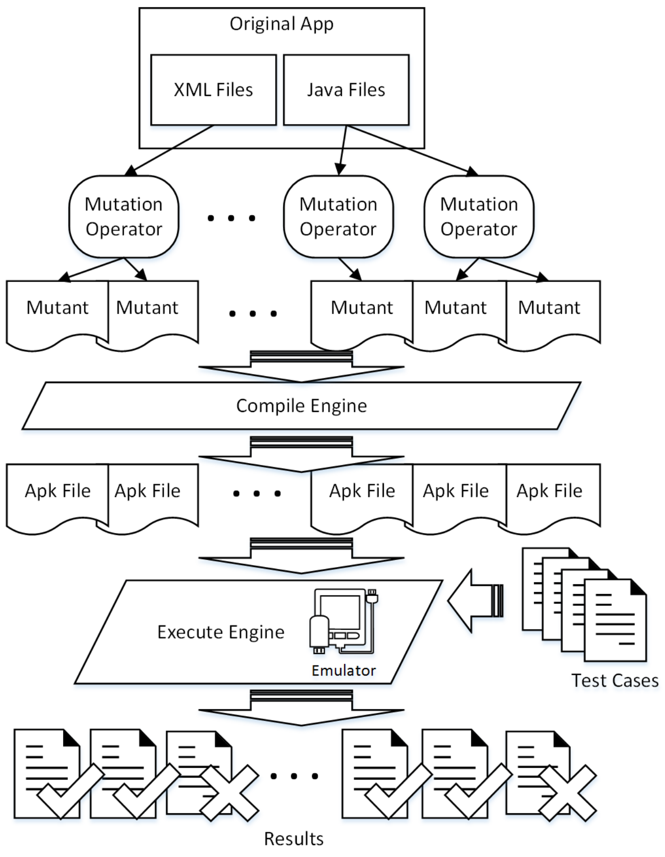

Fig. 6. Performing mutation analysis on Android apps

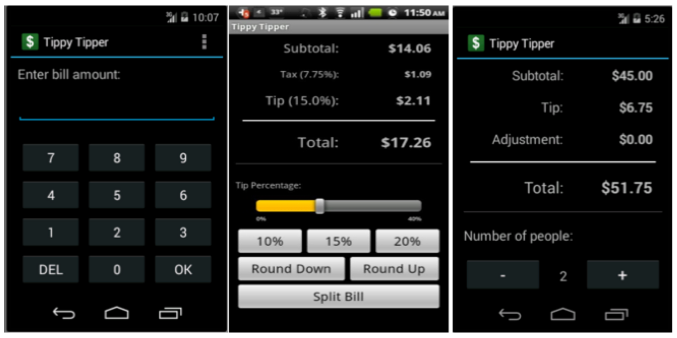

Fig. 7. Empirical Subject

the future.

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 V, the Android mutation operators do not seem to create many equivalent mutants.

V. Evaluation

To evaluate our proposed approach, we developed a new mutation analysis tool to generate mutants, compile the APK files, install the APK files into an emulator, execute tests against the mutants, and compute and report the final results. We present results of applying this tool to a small Android app.

A. Empirical Subject

TippyTipper [10] 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 who rated it. 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 7 illustrates three Activities of it: TippyTipper is on the left, the middle screen is SplitBill, and the rightmost one is Total.

We selected the main Activity TippyTipper and its corresponding XML layout file main.xml to test. The entire app has 12 Java classes distributed among five Java packages. It has eight XML layout files and one XML configuration file. We tested the main class, TippyTipper.java, which has 103 lines of code as calculated by Emma [50]. We determined that one line was dead code during testing. We also tested the XML layout file main.xml, which contained 93 lines as counted by our editor.

Class	Method Coverage	Block Coverage	Line Coverage
TippyTipper	82%	90%	85%

TABLE II
Structural coverage measures of test cases

B. Test Data Generation

We used the evolutionary algorithm-based tool EvoDroid [40] to generate test inputs. EvoDroid generated 744 test cases through multiple generations. We chose ten tests from the last generation that appeared to have good coverage. This study was not trying to measure the test generation technique, but to demonstrate feasibility of the mutation analysis approach, thus the particulars of the tests we generated were not important. The test cases generated by EvoDroid do not contain test oracles, so we added them by hand. We measured coverage by inserting assertions into the source before and after each action sent to the emulator. Table II shows that the test set covered 82% of the methods, 90% of the blocks, and 85% of the statements in the main Activity TippyTipper.java. According to the evaluation results of EvoDroid [40], the test set generated by EvoDroid is able to cover 82% of statements of the entire TippyTipper app. As a second step, we augmented the EvoDroid tests to achieve full statement coverage. We were able to do that with one additional test, designed by hand, thus achieving 100% statement code with the exception of one line of dade code.

C. Mutant Generation

Using the eight Android mutation operators, our mutation system tried to generate 287 mutants for TippyTipper.java and 13 for main.xml. 110 mutants could not be compiled into APK files (stillborn) and thus were not counted. As stated in Section II, some mutants are stillborn because of incorrect syntax. Another reason is that Android apps use integers as the identification for pre-defined resources and values that are saved in a separate file. Some mutation operators mutate the identification integers, which makes it impossible for Android to locate these pre-defined values, and further prevents APK files from being compiled. Thus 190 mutants were created and compiled into APK files. Our mutation system generated 105 mutants from the 19 method level operators borrowed from muJava [37]. Of the 190 valid mutants, 35 crashed immediately after launching, including 16 AOIU, 16 LOI, 1 ITR, and 2 SDL mutants. These trivial mutants are marked as killed by all tests. Three operators, ETR, IPR, and APD did not create any mutants for TippyTipper.

On a MacBook Pro with a 2.6 GHz Intel i7 processor and 16 GB memory, generating a mutant and compiling it as an APK file took up to two seconds.

D. Empirical results on TippyTipper

Table III summarizes results from running the new Android mutants on the app. 57 of 85 Android mutants were killed by the automatically-generated test set for a mutation score of 67.06%. Based on our hand analysis, the Android mutation operators did not generate any equivalent mutants.

Operators	Total Mutants	Killed Mutants	Equivalent Mutants	Mutation Scores
ECR	66	45	0	68.18%
MDL	1	1	0	100.00%
ITR	5	5	0	100.00%
BWD	12	6	0	50.00%
TWD	1	0	0	0.00%
Total	**85**	**57**	**0**	**67.06%**

TABLE III
EMPIRICAL RESULTS FOR THE ANDROID MUTATION OPERATORS

Operators	Total Mutants	Killed Mutants	Equivalent Mutants	Mutation Scores
AOIS	8	0	4	0.00%
AOIU	20	17	0	85.00%
AORB	8	0	0	0.00%
CDL	2	0	0	0.00%
LOI	18	17	0	94.44%
ODL	4	0	0	0.00%
SDL	43	21	0	48.84%
VDL	2	0	0	0.00%
Total	**105**	**55**	**4**	**54.46%**

TABLE IV
EMPIRICAL RESULTS FOR THE TRADITIONAL MUTATION OPERATORS

On the same MacBook Pro and an emulator with Android 4.4.2, installing one mutant APK file in the emulator and executing it against 11 test cases took up to eight minutes. The entire execution cost nearly 20 hours.

Table IV summarizes results from running the traditional mutants on the app. 55 of 105 traditional mutants were killed. Four AOIS mutants were easily hand-identified to be equivalent, as they conducted post increment/decrement after a value was returned. So the mutation score is 54.46%.

Table V shows the results from the test set after it was augmented to reach 100% statement coverage. This table combines both types of mutants, Android and traditional. This improved test set killed 155 of 190 mutants in total, for a mutation score of 83.33%.

Operators	Total Mutants	Killed Mutants	Equivalent Mutants	Mutation Scores
Android Mutants				
ECR	66	66	0	100.00%
MDL	1	1	0	100.00%
ITR	5	5	0	100.00%
BWD	12	12	0	100.00%
TWD	1	0	0	0.00%
Traditional Mutants				
AOIS	8	0	4	0.00%
AOIU	20	18	0	90.00%
AORB	8	0	0	0.00%
CDL	2	0	0	0.00%
LOI	18	18	0	100.00%
ODL	4	0	0	0.00%
SDL	43	35	0	81.40%
VDL	2	0	0	0.00%
Total	**190**	**155**	**4**	**83.33%**

TABLE V
EMPIRICAL RESULTS FROM THE 100% STATEMENT COVERAGE TEST SET

E. Discussion

Figure 8 compares results between two test sets. Once the test set was augmented to achieve 100% statement coverage, the mutation score on the Android mutants was very high. This set killed all but one Android mutant. This could be interpreted to mean that statement coverage is a very effective way to test Android apps, which from the field's long term experience with statement coverage seems unlikely. A more likely interpretation is that the current collection of Android mutants is not strong enough. We return to this point in future work.

VI. RELATED WORK

This section describes relevant research in two areas: Android testing and mutation testing.

A. Android Testing

Android's development environment includes its own test framework [3], which extends the ubiquitous JUnit. Many testers also use Robotium [9], a powerful open source Android test automation framework, to write unit, system, and user acceptance tests. It enables people to write tests with very little knowledge of the implementation details of the app under test. Because it provides APIs that directly interact with Android GUI components by run-time binding, 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. Another framework for Android apps is Robolectric [8], which splits tests from the emulator, making it possible to run tests by directly referencing the Android library files. All three 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. [13], [14] presented an approach that starts with random inputs, then uses a code-crawling algorithm to generate test cases. Hu and Neamtiu [29] generated GUI test inputs randomly, and executed them with Android Monkey. The tool Dynodroid [38] creates random values and sequences of events, and uses heuristics to increase the speed of Android Monkey.

Some researchers have extracted models to test Android apps. *ORBIT* [54] creates a GUI model of the app and then generates tests. A^3E [18] 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.

Jensen et al. [30] 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.

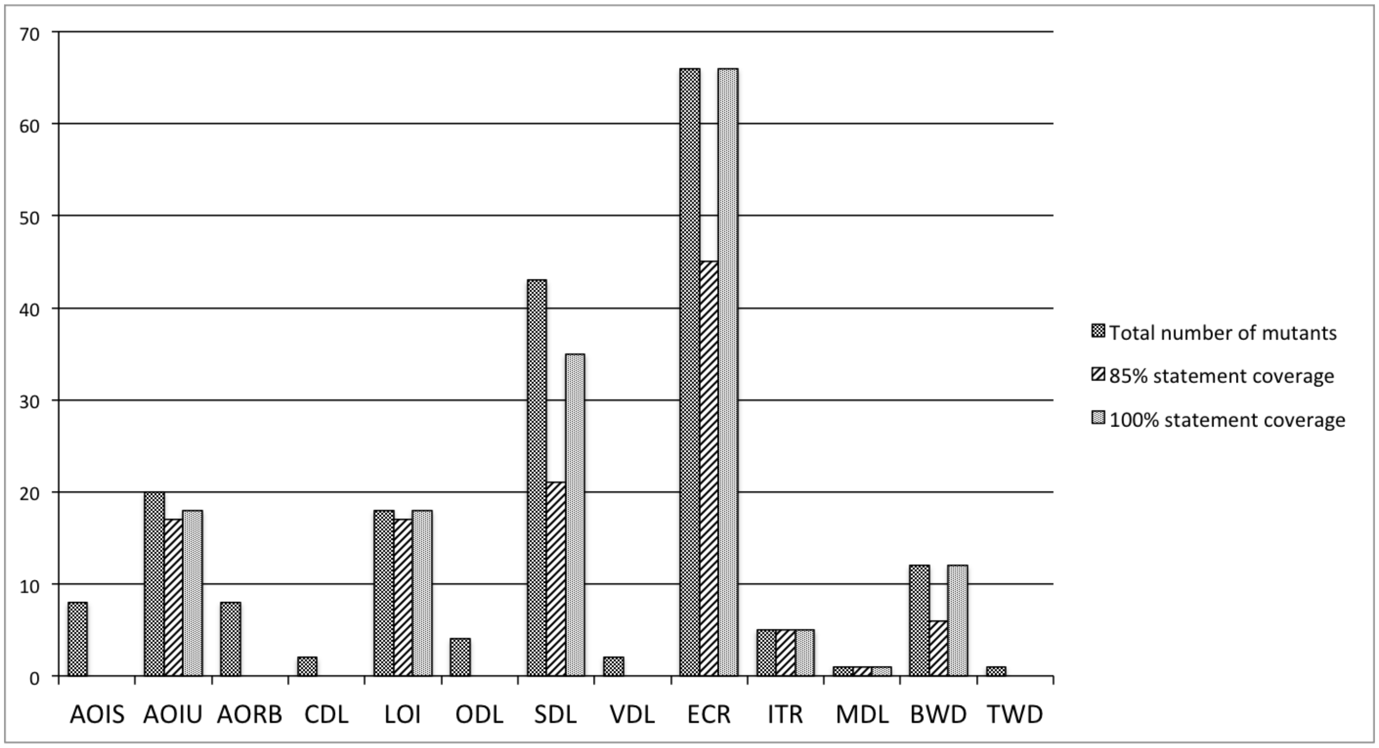

Fig. 8. Results Comparison between 85% Statement Coverage and 100% Statement Coverage

Anand et al. [17] used dynamic symbolic execution [34], [46] in the form of concolic testing [27], to test an Android library. Their testing used pixel coordinates to identify valid GUI events. Finally, several papers used evolutionary search [39], [40] and symbolic execution [42] to test Android apps. They focused on generating inputs for GUI testing of Android apps, instead of using test criteria.

B. Mutation Testing

Mutation testing [21], [22] makes syntactic changes to an original program (mutants), then challenges the tester to design tests to *kill* the mutants by finding tests that cause each mutant to behave differently from the original. Mutation analysis can also be used to evaluate the strength a given test set or to guide testers to design effective tests. Mutation testing has been applied to many languages, including Fortran 77 [22], [33], C [19], Java [32], [37], Javascript [41], and web applications [49]. To our knowledge, mutation has not been applied to mobile apps.

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, *selective mutation* was proposed by Wong and Mathur by only choosing a subset of mutation operators [52], [53]. The muJava tool selects 15 operators to preserve almost the same test coverage as non-selected mutation [37].

Offutt and Xu approached the problem of input data validation for web services by designing mutation operators that modified XML schemas [48]. 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.

VII. Conclusion and Future Work

This paper proposes an innovative approach to conduct mutation analysis for Android apps. We defined new mutation operators specific to Android apps, implemented them in a mutation analysis tool, and conducted a preliminary experiment with a simple subject. The results show that mutation testing can fruitfully 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.

While promising, the work done so far requires significant additional research.

1) We are unaware of any comprehensive fault models for Android apps. Instead, we relied on Android syntax to define the new Android mutation operators in this paper. An 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.

2) In this paper, we defined eight new Android mutation operators, from mutating Java source code, XML layout, to Android permissions. However, there are aspects of Android apps that we have not yet considered. For instance, one important distinct characteristic of mobile apps is that they are context-aware. For example, location-aware apps behave differently when the owner of the phone is moving in a vehicle vs. 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.

   As another example, most mobile devices usually have two orientations: landscape and portrait. Mobile applications should be adaptive to both orientations, as well as different screen resolutions. Our mutation system should definitely incorporate screen size and orientation. This is challenging because most issues with screen size and orientation show up as usability problems rather than functional failures.
3) Our Android mutation tool is very much a prototype and requires additional development, such as designing better mutation operators and algorithms. In particular, we need to make our tool generate fewer stillborn mutants, fewer crashed mutants, and more hard to kill mutants. Also, we would like to employ external well-established frameworks, such as Xposed [11], to facilitate our mutation analysis.
4) The experiment we discussed in this paper is quite limited. For the subject, we selected a single main activity in a single Android app. Clearly, we plan to do more experimentation with more apps.
5) The cost of mutation testing Android apps is especially expensive due to the slow speed of Android test execution with Robotium. Our small experiment required 20 hours for a single iteration.

   One possible solution is to evaluate mutants in parallel. A more interesting approach would be to choose mutants more carefully. Work in general program mutation suggests that only a small number of generated mutants are necessary [15]; this result may extend to Android mutation as well.

ACKNOWLEDGMENT

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

REFERENCES

[1] Android developers guide. Last access January 2015. [Online]. Available: http://developer.android.com/guide/topics/fundamentals.html

[2] Android intent. Last access January 2015. [Online]. Available: http://developer.android.com/reference/android/content/Intent.html

[3] Android testing framework. Last access January 2015. [Online]. Available: http://developer.android.com/guide/topics/testing/

[4] ART and Dalvik. Last access March 2015. [Online]. Available: https://source.android.com/devices/tech/dalvik/index.html

[5] Dalvik - code and documentation from Android's VM team. Last access January 2015. [Online]. Available: http://code.google.com/p/dalvik/

[6] Google play. Last access January 2015. [Online]. Available: https://play.google.com/store

[7] JUnit. Last access January 2015. [Online]. Available: http://junit.org

[8] Robolectric. Last access January 2015. [Online]. Available: https://github.com/robolectric/robolectric

[9] Robotium. [Online]. Available: http://code.google.com/p/robotium/

[10] TippyTipper. Last access January 2015. [Online]. Available: https://code.google.com/p/tippytipper

[11] Xposed Framework. Last access March 2015. [Online]. Available: http://repo.xposed.info

[12] H. Agrawal *et al.*, "Design of mutant operators for the C programming language," Software Engineering Research Center, Purdue University, West Lafayette IN, Technical Report SERC-TR-41-P, March 1989.

[13] D. Amalfitano, A. Fasolino, and P. Tramontana, "A GUI crawling-based technique for Android mobile application testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, March 2011, pp. 252–261.

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and 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*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261.

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

[16] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008, iSBN 978-0-521-88038-1.

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

[18] T. Azim and 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*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 641–660.

[19] M. E. Delamaro and 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, July 1996, pp. 79–95.

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

[21] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[22] R. A. DeMillo and J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

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

[24] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *5th IEEE International Symposium on Software Reliability Engineering (ISSRE 94)*, Monterey, CA, November 1994, pp. 220–229.

[25] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and 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, October 1995, pp. 329–337.

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

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

[28] R. Hierons and M. Merayo, “Mutation testing from probabilistic finite state machines,” in *Third Workshop on Mutation Analysis (IEEE Mutation 2007)*, Windsor, UK, September 2007, pp. 141–150.

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

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

[31] S. Kim, J. A. Clark, and J. A. McDermid, “Investigating the applicability of traditional test adequacy criteria for object-oriented programs,” in *Proceedings of ObjectDays 2000*, October 2000.

[32] ——, “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, October 2000, pp. 4–100, Wiley’s Software Testing, Verification, and Reliability, December 2001.

[33] K. N. King and J. Offutt, “A Fortran language system for mutation-based software testing,” *Software-Practice and Experience*, vol. 21, no. 7, pp. 685–718, July 1991.

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

[35] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes, “Testing aspect-oriented programming pointcut descriptors,” in *Proceedings of the 2nd workshop on testing aspect-oriented programs*. ACM, 2006, pp. 33–38.

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

[37] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, “MuJava : An automated class mutation system,” *Software Testing, Verification, and Reliability, Wiley*, vol. 15, no. 2, pp. 97–133, June 2005.

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

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

[40] R. Mahmood, N. Mirzaei, and S. Malek, “Evodroid: Segmented evolutionary testing of Android apps,” in *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE ’14. Hong Kong, China: ACM, November 2014.

[41] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, “Efficient JavaScript mutation testing,” in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 74–83.

[42] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, “Testing Android apps through symbolic execution,” *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012.

[43] L. J. Morell, “A theory of fault-based testing,” *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, August 1990.

[44] R. Nilsson, J. Offutt, and J. Mellin, “Test case generation for mutation-based testing of timeliness,” in *Proceedings of the 2nd International Workshop on Model Based Testing*, Vienna, Austria, March 2006, pp. 102–121.

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

[46] J. Offutt, Z. Jin, and J. Pan, “The dynamic domain reduction approach to test data generation,” *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–193, January 1999.

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

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

[49] U. Praphamontripong and J. Offutt, “Applying mutation testing to web applications,” in *Sixth Workshop on Mutation Analysis (IEEE Mutation 2010)*, Paris, France, April 2010.

[50] V. Roubtsov, “Emma,” Online, 2006, http://emma.sourceforge.net/, last access January 2015.

[51] 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, September 2007, pp. 151–160.

[52] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Constrained mutation in C programs,” in *Proceedings of the 8th Brazilian Symposium on Software Engineering*, Curitiba, Brazil, October 1994, pp. 439–452.

[53] W. E. Wong and A. P. Mathur, “Reducing the cost of mutation testing: An empirical study,” *Journal of Systems and Software, Elsevier*, vol. 31, no. 3, pp. 185–196, December 1995.

[54] W. Yang, M. R. Prasad, and 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*, ser. FASE’13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 250–265.