2017 IEEE International Conference on Software Quality, Reliability and Security

Is Mutation Analysis Effective at Testing Android Apps?

Lin Deng, Jeff Offutt, and David Samudio
Department of Computer Science
George Mason University, Fairfax, Virginia, USA
{ldeng2, offutt, dgonza10}@gmu.edu

***Abstract*—Not only is Android the most widely used mobile operating system, more apps have been released and downloaded for Android than for any other OS. However, quality is an ongoing problem, with many apps being released with faults, sometimes serious faults. Because the structure of mobile app software differs from other types of software, testing is difficult and traditional methods do not work. Thus we need different approaches to test mobile apps. In this paper, we identify challenges in testing Android apps, and categorize common faults according to fault studies. Then, we present a way to apply mutation testing to Android apps. Additionally, this paper presents results from two empirical studies on fault detection effectiveness using open-source Android applications: one for Android mutation testing, and another for four existing Android testing techniques. The studies use naturally occurring faults as well as crowdsourced faults introduced by experienced Android developers. Our results indicate that Android mutation testing is effective at detecting faults.**

***Index Terms*—Android, Software Testing, Mutation Testing, Empirical Evaluation, Crowdsourcing**

I. Introduction

Mobile applications are software programs that are installed and executed on mobile devices. Users currently spend more time using mobile apps than any other type of software program [18]. The total number of Android apps available in the Google Play store exceeds 2.8 million [4]. However, the quality of Android apps is a serious and growing problem. Research shows that many apps contain significant faults, resulting in regular failures during use [14]. A professional Android analysis website [4] labeled 13% of all the Android apps on the market as "low quality apps." Although an exact number is unknown, the percentage of Android apps with significant faults is much higher. Android apps have unique characteristics and novel features [25], which brings new challenges to Android testers.

Our previous papers [12], [13] took the first steps toward applying mutation testing to Android apps by focusing on the unique characteristics of Android apps. Those papers did not evaluate the fault detection effectiveness of Android mutation testing. Additionally, our previous work did not consider several unique characteristics of Android apps, such as the Service component, some frequently used GUI widgets (e.g., TextView), high energy consumption issues, and context-aware issues (e.g., changing location). This paper extends our previous work by designing and implementing six novel Android mutation operators to address the additional challenges in testing Android apps, and evaluating the fault detection effectiveness of Android mutation testing and four other Android testing techniques. The research used two sets of faults: naturally occurring faults mined from our subject apps' source code repositories and hand-seeded faults collected from experienced Android developers. The evaluation results show that Android mutation testing is very effective at detecting both kinds of faults. Because mutation testing has relatively higher costs than other testing techniques, this paper does not directly compare Android mutation testing with the existing Android testing techniques, but presents fault detection effectiveness of different techniques.

In summary, the contributions of this paper are:

1) We evaluated the fault detection effectiveness of Android mutation testing and four existing Android testing techniques.
2) We collected naturally occurring faults from nine subject apps by mining their source code repositories, and invited Android developers online to hand-seed faults[1] into the subjects. These faults can be reused as a benchmark to evaluate other Android testing techniques.
3) We added six new Android mutation operators to our 11 existing Android mutation operators to address additional testing challenges, and included all these 17 operators in our evaluation.
4) We adopted a new crowdsourcing approach to recruit participants in our empirical study. A rigorous selection process ensured that our participants were competent to design and insert faults for the experiment.

This paper is organized as follows. Section II introduces background on Android apps, mutation testing, and crowdsourcing. Section III describes challenges in testing Android apps and common faults found in Android app development. Section IV defines six new Android mutation operators. Section V describes the experiment used to evaluate the fault detection effectiveness, and Section VI presents and analyzes the experiment results. Section VII gives an overview of related research, and the paper concludes and suggests future work in Section IX.

II. Background

This section provides background on Android apps, mutation testing, and crowdsourcing in software engineering.

A. Android Applications

Each Android app must contain one or more of the following components: *Activity*, *Service*, *Content Provider*, and

[1] https://cs.gmu.edu/~ldeng2/AndroidFaults.html

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.19

Broadcast Receiver. They are written in Java, and developers can extend them to provide their own implementation based on requirements. Activities construct the presentation layer of an app as a Graphical User Interface (GUI). Developers separate the visual structure from the behavior by using XML to declare the app's layout. Services do not have a GUI, and run in the background without direct user interaction. Thus, they are usually used to perform long term tasks. A Content Provider supplies and manages structured data to other apps. These data are stored in file systems or databases. Apps use Broadcast Receiver to subscribe to intents broadcast by the Android system or other apps.

B. Mutation Testing

Mutation testing dates back to 1978 [11]. It is a syntax-based testing technique that efficiently helps testers design high quality tests, and is widely considered to be an exceptionally effective criterion for both designing and evaluating tests [7]. The general mutation testing process includes the following steps. First, the original program under test P is modified to create a set of different new versions, P', called *mutants*. Each mutant encodes a single syntactic change from P. The rules that specify how to create mutants are called *mutation operators*. The tester then creates and executes test cases T against both the original program P and all mutated versions P'. If a test t in the test set T causes different outputs on the original P and one mutant p in P', the test t is said to *kill* the mutant. The percentage of mutants killed by t is called the *mutation score*. If a mutant always has the same output as the original program P, it is called *equivalent*. Once all the mutants are killed, T has 100% mutation score and is called *mutation adequate*. Mutation adequate test sets have been found to be very effective at finding faults.

C. Crowdsourcing in Software Engineering

Crowdsourcing is the act of using an open call to recruit a group of professionals and assign them tasks [24]. LaToza and van der Hoek [19] defined essential dimensions to classify crowdsourcing approaches, and illustrated three crowdsourcing models: *peer production*, *competitions*, and *microtasking*. This paper uses the *microtasking* model to collect crowdsourced faults, that is, hand-seeding faults is considered to be a microtask. To our knowledge, this is the first attempt to use crowdsourcing to create faults in software testing empirical studies.

III. Challenges in Testing Android Apps

Even though a part of Android apps' source code is written in Java, they still have unique characteristics and features beyond traditional software. In particular, testing Android apps is different from testing traditional Java programs. We identify seven unique challenges and three types of common faults that should be appropriately addressed when testing Android apps.

1) Unique lifecycles of Android components: All major components of Android apps need to behave according to a pre-specified lifecycle [1]. Inappropriately handling a component's lifecycle is likely to cause unexpected issues [9]. For example, when developing gaming apps, the flow of continuity is critical to users. When an incoming call interrupts the user from gaming, the app should be able to properly handle the lifecycle to pause the game process and store any temporary data. Once the user finishes the call, the game process should be recovered from where it was stopped.

2) Intensive use of eXtensible Markup Language (XML) files: Android apps depend on XML files for program configuration, user interface (layout) specification, temporary data storage, etc. Traditional Java programs with Graphical User Interfaces (GUIs) rarely use XML files. So traditional testing techniques do not target source code other than Java. Additionally, there is no test coverage criterion to measure coverage for XML files. Obviously, not testing XML files of Android apps may result in unexpected failures.

3) Context-aware characteristics: Android apps are context-aware because they receive input from the physical environment through sensors, such as accelerometer, ambient temperature sensor, and magnetic field sensor. These types of data are not directly fed by users' inputs, but from sensors in the device itself. These event notifications are inputs as well, and must be modeled as part of a test. However, existing test techniques do not consider these types of inputs.

4) Two types of screen orientation: The ability to change orientation is a key characteristic of mobile devices. Almost every device has two types of screen orientation: landscape and portrait. Many Android apps have different user interfaces to adapt to the orientation change event. Testing Android apps must consider this unique feature of orientation changing, because it is highly likely to cause failures such as immediately crashing after switching the orientation [15].

5) Event-based programs and system buttons: Android apps are event-based programs, as their execution flows are dependent on user actions, such as clicking, tapping, and dragging. As these events are handled by different kinds of event handlers, improperly designed event handlers could lead to incorrect behaviors. In addition, every Android device is equipped with three system buttons: *Back, Home,* and *Recents*. Because the system buttons interrupt the usual execution flow and are usually not on the commonly expected execution flow, many testers forget to test the impact of the system buttons.

6) Varied network connections: Most Android devices are equipped with multiple forms of network connections, most commonly cellular data and Wi-Fi. Whenever a Wi-Fi connection is available, the Android system will first attempt to transmit data through Wi-Fi, as Wi-Fi connection uses less energy, is faster, and costs less. If there is no Wi-Fi, the Android system will try to switch to a cellular data connection, which is more expensive and uses more battery. This switching can cause problems in different scenarios. The network a device switches to may have difficulty resuming the connection and may fail to continue unfinished tasks. Developers may overlook this situation, leaving it to users to remember to remain tethered to Wi-Fi when an important app is running.

7) Limited battery life: Mobile devices have to rely on limited battery power. Therefore, developers should, but sometimes do not, take battery usage into account when implementing their apps. Researchers found that certain programming practices increase energy consumption on Android apps [16],

TABLE I: Android Mutation Operators

Category	Android Mutation Operator
Event-based	Intent Payload Replacement (IPR)
	Intent Target Replacement (ITR)
	OnClick Event Replacement (ECR)
	OnTouch Event Replacement (ETR)
Component Lifecycle	Activity Lifecycle Method Deletion (MDL)
	Service Lifecycle Method Deletion (SMDL)
XML-related	Button Widget Deletion (BWD)
	EditText Widget Deletion (TWD)
	Activity Permission Deletion (APD)
	Button Widget Switch (BWS)
	TextView Deletion (TVD)
Common Faults	Fail on Null (FON)
	Orientation Lock (ORL)
	Fail on Back (FOB)
Context-aware	**Location Modification (LCM)**
Energy-related	**WakeLock Release Deletion (WRD)**
Network-related	**Wi-Fi Connection Disabling (WCD)**

[28]. These are called *energy bugs*. Even though energy bugs do not affect the functionality of an app, they can shorten the availability of the system by running down the battery.

8) Common Faults in Android Apps: Mutation operators are based, in part, on common faults in the domain being considered. Our study on GitHub repositories revealed three types of common faults in Android app development. First, many developers forget to check whether an object is null before accessing it. Thus, *NullPointerException* is the most common exception Android apps throw. Our study shows that, in a total of 80 patches collected from the repository of DAVdroid [3], 52 fixed null-checking faults. Second, switching the screen orientation may result in failures, such as resetting on screen data, deforming GUI widgets, and crashing. Third, we have identified the testing challenge of the *Back* button in Section III-5. A common failure is crashing when the *Back* button is clicked. Zaeem et al. [31] conducted a fault study and categorized faults specific to Android apps. In addition to incorrect application logic faults, common faults include uncaught exceptions, visual appearance, screen rotations, website connections, and the Activity lifecycle. These common faults are included in the testing challenges identified in this paper.

IV. Android Mutation Operators

The unique testing challenges in Section III indicate that a sophisticated testing technique that can provide comprehensive testing for Android apps is still needed. Although the focus in this paper is on the experimental study in Section V, our previous research [12], [13] did not consider several challenges, thus this paper introduces six new Android mutation operators to address all the identified challenges of testing Android apps. Table I lists all Android mutation operators, highlighting the operators new to this paper. This section describes the six new operators; the others are described in the previous papers.

A) Service Component Mutation Operator

Service Lifecycle Method Deletion (SMDL): As discussed in Section III-1, inappropriately handling the unique lifecycles of Android components is a common mistake in Android app development. Services behave according to another type of pre-defined lifecycle that is not the same as the lifecycle of Activity. Figure 1 shows the lifecycle of two different types of Service as a collection of event methods and states.

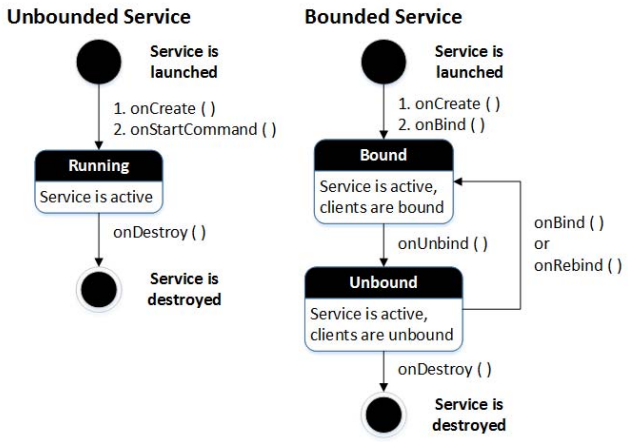

Fig. 1: Lifecycle of a Service in Android Apps

An unbounded Service is launched after the Android system calls its *onCreate*() and *onStartCommand*() methods, usually when other client components of the app request the Service. Then, the Service stays in the background and performs its job. Once the Service finishes its task, it can stop itself or be stopped by its client. The Android system calls the *onDestroy*() method to terminate the Service. A bounded Service is started once a client component asks to bind to it. The Android system calls its *onCreate*() and *onBind*() methods to launch it. The Service can accept binding requests from multiple clients at the same time. If the Service is purely a bounded Service, i.e., launched with a client's binding request, after all clients unbind from the Service, the Android system calls the *onUnbind*() and the *onDestroy*() methods to stop the Service.

Correctly implementing Services according to the lifecycle is critical to the communication between Service and other components, and further leads to the smooth execution of Android apps, especially for apps that rely heavily on the correct running of Services, such as email clients and alarm clocks. We have identified faults in Services of an alarm clock app that cause the alarm to sound at a wrong time, or crash.

SMDL deletes each lifecycle method in Services, including *onCreate*(), *onStartCommand*(), *onBind*(), *onRebind*(), *onUnbind*(), and *onDestroy*(). To kill an SMDL mutant, the tester must design a test that ensures the Service works as expected.

B) XML-related Mutation Operator

TextView Deletion (TVD): As mentioned in Section III-2, XML files form one of the key challenges in testing Android apps. This section presents an XML mutation operator specific to one type of GUI widget, TextView. Android apps use TextView to display text to users. Unlike EditText, TextView usually cannot be edited by users. The left screenshot in Figure 2 is from an Android app. "Subtotal," "Tip (15.0%)," "Total," and their numbers are all TextView widgets. As can be seen, TextView widgets usually do not associate with any user events, nor require any event handlers from the implementation of the app. However, TextView is widely used by developers

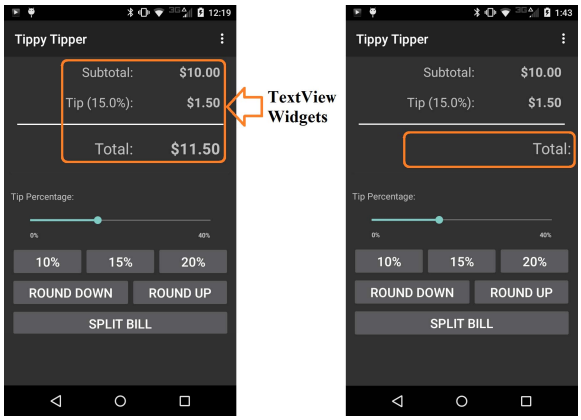

Fig. 2: An Example of TextView Widgets and TVD Mutant

to present essential information.

TVD deletes TextView widgets from screens one at a time. The right side of Figure 2 shows an example TVD mutant in which the TextView Widget of the total amount is deleted. To kill a TVD mutant, the tester needs to design a test to check whether the total amount is correctly displayed.

C) Common Faults Mutation Operator

Fail on Back (FOB): Section III-5 introduced the testing challenges caused by the Android system buttons. At the application level, the *Back* button is the most impactful. Unlike the *Home* and *Recents* buttons that pause or terminate the app, the *Back* button lets users move back to previous screens. Many testers might overlook its impact. A common failure is the app crashing when the *Back* button is clicked. FOB injects a "Fail on Back" event handler into every Activity. To kill FOB mutants, testers need to design tests that press the *Back* button at least once at every Activity.

D) Context-Aware Mutation Operator

Location Modification (LCM): As stated in Section III-3, Android apps are context-aware, a unique feature of mobile apps. Location data is the most frequently and widely used context input data. Other context data are either managed by the Android operating system (such as ambient light and temperature), or are rarely used by Android apps (such as gravity and pressure). Consequently, we first consider location data when designing mutation operators. LCM injects code to modify the attribute values of every location variable, in terms of its latitude, longitude, altitude, and speed, with pre-defined values. Specifically, LCM changes latitude and longitude values by adding and deducting one degree, which is equivalent to moving the device roughly 115 kilometers. For altitude values, LCM elevates and lowers them by adding and deducting one meter. For speed values, LCM accelerates and decelerates them by adding and deducting one meter per second. To kill an LCM mutant, the tester needs to design tests to ensure the app behaves as expected at different locations.

E) Energy-Related Mutation Operator

WakeLock Release Deletion (WRD): Section III-7 introduced the testing challenge regarding limited battery power of Android devices, and energy bugs. Guo et al. found that some energy bugs were caused by resource leaks, i.e., resources acquired too early or released too late that deplete a device's battery. One typical resource leak is inappropriately using *wake locks* [16]. A *wake lock* is a mechanism used by the Android system to keep devices from going into sleep mode. Figure 3 compares two energy consumption diagrams from the same app. Initially, the app has an *energy bug* (top), which causes a performance failure, hence it consumes an unexpectedly large amount of energy after its state transits from active to idle. An app should use resources when active, but not when idle. These states differ from background and foreground states of Android components because an app can be active while in the background (e.g., media players). After fixing the bug, the app's energy consumption drops when entering the Idle state.

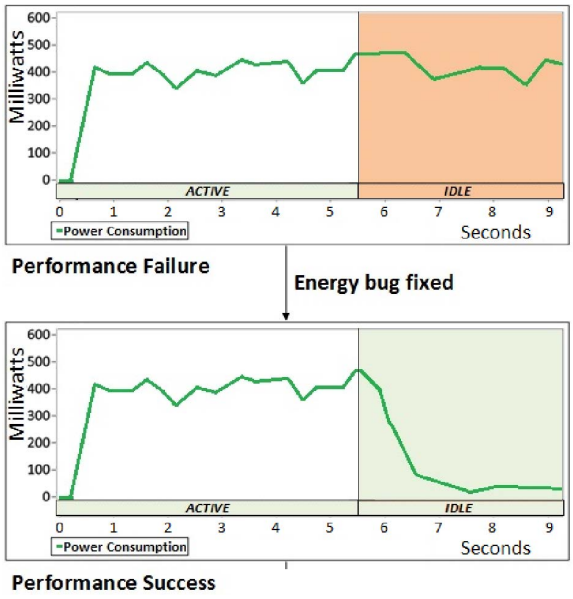

Fig. 3: An Example Energy Bug

Android apps request wake locks with the *acquire*() method. When the wake lock is not needed, the app should call the *release*() method to release it. In 2014, Samudio designed and implemented an automated Android energy inspection tool [29] to detect and correct *energy bugs*. The tool checks for inappropriately acquired and released wake locks and other resources, offers to fix them, and also presents visualizations of their energy consumption, both statically and dynamically. We used this idea to design the WRD mutation operator, which deletes each call to *release*() to force the app not to release the wake lock. It mimics a typical *energy bug*, which commonly happens when the app retains a resource during an idle state.

Testers can kill WRD mutants in one of two ways. First, using the tool *dumpsys* in Android SDK, testers can identify active wake locks after the app under test has been closed. Figure 4 shows part of the output from the *dumpsys* tool after launching the subject JustSit, that is, when the app is in its active state. According to the output, JustSit has an active wake lock. If the wake lock is handled properly, the app should release it successfully after it transits to idle. If the wake lock is still active in idle state, the WRD mutant is

killed. Alternatively, testers can use external tools to profile

```
Wake Locks: size = 3
  PARTIAL_WAKE_LOCK          'WakeLock.Local' (uid=10145
  SCREEN_DIM_WAKE_LOCK       'JustSit' ON_AFTER_RELEASE
  PARTIAL_WAKE_LOCK          'AudioMix' (uid=1013, pid=0
```

Fig. 4: Identifying Wake Locks in the Android System

and compare the energy consumption of the original app and the mutants. If testers can identify irregular battery drainage, the mutants are killed.

F) Network-related Mutation Operator

Wi-Fi Connection Disabling (WCD): The challenge discussed in Section III-6 urges testers to consider varied network connections when designing tests. WCD inserts a special piece of code into every Activity to disable the Wi-Fi connection when the Activity is launched. WCD mutants mimic the scenario that the device drops Wi-Fi connection and is forced to switch to another connection. To kill a WCD mutant, the tester must design tests that test different network scenarios.

V. Experiment

We built our Android mutation testing tool by extending muJava [22]. We implemented all the mutation operators presented in Section IV, as well as the operators from our previous work [13]. We carried out a two-part experiment to investigate the following research questions:

RQ1: How effective is Android mutation analysis in testing Android apps? Specifically, how many faults can be detected by mutation-generated tests?

RQ2: How effectively do four other Android testing techniques test Android apps? Specifically, with the same set of faults, how many of them can be detected by four other Android testing techniques?

We used both naturally occurring and hand-seeded faults. The results are presented separately to check whether the results are significantly different and because we had an order of magnitude more hand-seeded faults. Note that these comparisons are based on effectiveness (the ability to find faults), and do not account for cost. Android mutation testing is quite a bit more costly than the other techniques. However, the other techniques are the current state-of-the-art and no structural-based techniques or tools for testing mobile apps are available to compare with.

A. Experiment Subjects

Table II summarizes the nine subjects used in our experiment. We calculated the Lines of Code (LOC) in the Java files with Metrics [5], and the numbers of XML nodes in the XML layout and manifest (configuration) files with our own XML parser program. Jamendo is the largest app, with 8287 LOC and 380 XML nodes. The smallest is Tipster, with 222 LOC and 37 XML nodes. We used the 19 method-level Java mutation operators from muJava [22] to generate 8,940 muJava mutants, and used the 17 Android mutation operators to generate 4,739 Android mutants for Activities, Services, XML layout files, and configuration files of the subjects. The number of muJava mutants ranges from 306 for WorldClock to 2,264 for Jamendo, and the number of Android mutants ranges from 130 for Tipster to 1,417 for Jamendo.

TABLE II: Details of Experiment Subjects

Apps	LOC	XML Nodes (Layout and (Manifest)	muJava Mutants	Android Mutants
AlarmKlock	3899	124	1223	822
AndroidomaticKeyer	2203	72	1517	869
Jamendo	8287	380	2264	1417
JustSit	479	45	562	331
Lolcat Builder	851	25	581	388
MunchLife	332	31	581	170
TippyTipper	1521	153	1579	434
Tipster	222	37	327	130
WorldClock	1436	49	306	178
Total	**19,230**	**916**	**8,940**	**4,739**

B. Collecting Naturally Occurring Faults

All nine subject apps have their own GitHub repositories, which we used to collect naturally occurring faults by searching their committed revisions. They also all have issue tracking systems, so that we can retrieve information about the faults. We examined every commit in the repositories looking for the keywords *fix*, *fault*, *bug*, *issue*, and *incorrect*. In each commit related to a fault, we collected three types of data: the source code of the faulty version (the prior version), the source code of the fixed version, and the description of the commit. Some developers linked their commits with the bug reports in their app's issue tracking system. After that, we examined every commit and discarded commits that did not fix a fault. Several commits were labeled with the keywords "fix" or "issue," but did not fix a fault, or just changed the usability or performance. We also did not consider commits that fixed typos in apps.

We verified every fault to check whether it could be reproduced in our experimental environment. Some faults could not be reproduced because either the faulty version or the fixed version used obsolete APIs. Other faults were specific to only one brand or one model of devices, or specific to only one version of Android system. Some faults are based on external systems that we could not practically emulate, such as an email client app that cannot delete emails from a specific user's server. We discarded all faults that could not be reproduced. Two subjects, Tipster and Lolcat Builder, do not have commits explicitly related to fixed faults. In total, we collected 51 commits with fault fixing activities, in which 25 faults could be reproduced in our experiment environment, and 26 faults were discarded. We used these 25 reproducible naturally occurring faults in our experiment.

C. Hand-seeding Faults

We recruited anonymous Android developers from a crowdsourcing website[2] to seed faults into our nine experiment subject apps. We required our candidates to have established records in Android app development, to have previous projects finished on time and on budget with a 5-star rating, and to be able to communicate in English. We also verified our candidates' identities by phone, email, or Facebook. After this initial verification, we interviewed each applicant to determine his or her knowledge and experience in developing and testing Android apps. Even though all the freelancers were experienced Android developers, they were not familiar with seeding

[2]www.Freelancer.com

faults into software programs. One freelancer kept trying to identify possible faults in our subjects, and we eventually had to replace him with another candidate. We only selected applicants who understood the project in general but **did not** know mutation testing. We did not give them any instructions or guidelines on how to seed faults. They were free to seed any faults based on their own developing and testing experience.

After this rigorous process, our crowd was composed of seven freelancers from four countries. We gave them $1 per each valid seeded fault and asked them to seed up to 200 faults in our nine subjects. We considered each microtask, hand-seeding a fault, to be valid if: (1) it changed the source code, including Java classes, XML files, or any other files; (2) the changed app compiled and executed in our experiment environment; (3) the changed app did not crash immediately after launching (otherwise it would be too trivial to use in the experiment); (4) the fault caused the subject app to behave differently from the original version; and (5) at least one test case must be able to observe it.

Overall, our crowd created 589 hand-seeded faults for the nine subject apps. One problem with not giving detailed directions about the type of faults needed is that the fault seeders quite naturally created faults that were actually mutants. Thus, we hand-inspected each fault, and eliminated faults that were also mutants to avoid biasing the results in favor of mutation. We were left with a total of 437 non-mutant hand-seeded faults to use in our experiment.

D. Other Android App Testing Techniques

We used the same faults to compare four Android testing techniques. Monkey [2] is officially provided by the Android SDK, and is widely used by Android developers. Monkey sends pseudo-random user events to the app under test. Many developers use Monkey to stress-test their apps. Dynodroid [23] is an open source project developed by the Android testing research community. Dynodroid views Android apps as event-driven programs and uses an approach called *observe-select-execute* to generate test inputs. After executing an event, it observes the new state of the app and selects a relevant event for the next execution. PUMA [17] is a dynamic analysis tool that enables scalable UI automation for Android apps. It incorporates Monkey and uses the event-driven programming methodology. A^{3}E [8] provides systematic exploration for Android apps. It does not require the source code of the app under test, but constructs a model of the app with transitions among Activities using static taint analysis algorithms. The model is then used to automatically explore the Activities in the app.

E. Experiment Procedure

We used four steps to evaluate the fault detection effectiveness of Android mutation testing:

1) We used our Android mutation testing tool to generate mutants for the nine experiment subject apps.
2) Tests were hand-designed by the first author to kill all non-equivalent mutants. Equivalent mutants were identified by hand during this step and eliminated.
3) We ran the mutation-adequate test set against each fault and recorded whether it was detected.
4) We then calculated the fault detection effectiveness of Android mutation testing for both kinds of Android faults.

Next, we evaluated the fault detection effectiveness of the four other Android testing techniques.

1) We used each tool to test each faulty app, and recorded whether the fault was detected.
2) We calculated, evaluated, and compared the fault detection effectiveness of the four Android testing techniques for both naturally occurring and hand-seeded Android faults.

We installed each faulty version of the apps into a clean emulator before testing so that each run was independent.

VI. Evaluation

This section presents the experimental results and discusses key findings.

A. Experiment Results

Table III shows the numbers of naturally occurring faults detected by Android mutation testing and the four existing Android testing techniques. Android mutation testing detected 18 out of 25. Monkey detected 6, Dynodroid detected 7, PUMA detected 3, and A^{3}E detected 1.

Table IV provides the results of hand-seeded faults detected by Android mutation testing and the four existing Android testing techniques. The mutation-based tests detected 360 of 437 hand-seeded faults. Monkey detected 130, Dynodroid detected 138, PUMA detected 121, and A^{3}E detected 79.

B. Discussion

For both groups of faults, the Android mutation tests found more faults than the other four Android testing techniques, and at a statistically significant level (more than twice as many). Because Android mutation testing addresses more unique characteristics and testing challenges of Android apps, and specifically targets faults that commonly occur during Android app programming, the results are not surprising. Inevitably, providing such comprehensive testing requires higher cost than automated testing techniques, in terms of time and effort in designing and executing tests. Therefore, a direct comparison of the fault detection ability of Android mutation testing and the other techniques must be tempered by the differences in basic strategy and cost. Android mutation testing is the first strong testing technique that enables Android app developers to design effective tests or evaluate the quality of existing tests.

All tools detected more hand-seeded faults than naturally occurring faults, although the difference was not statistically significant (less than 20% across the board). Thus, we cannot conclude that either population of faults led to different results.

According to our results, around 18% of hand-seeded faults and 28% of naturally occurring faults were not detected by the Android mutation-adequate tests. For example, only 50% of the hand-seeded faults in Lolcat Builder were detected by our tests. Lolcat Builder manipulates images, but our automated tests did not check images to decide whether tests failed or

TABLE III: Numbers and Percentages of Detected Naturally Occurring Faults

Apps	# Faults	Android Mutation	% by Mutation	Monkey	% by Monkey	Dynodroid	% by Dynodroid	PUMA	% by PUMA	A^{3}E	% by A^{3}E
AlarmKlock	7	6	85.71%	2	28.57%	4	57.14%	1	14.29%	1	14.29%
AndroidomaticKeyer	3	2	66.67%	1	33.33%	1	33.33%	0	0.00%	0	0.00%
Jamendo	6	4	66.67%	2	33.33%	2	33.33%	2	33.33%	0	0.00%
JustSit	1	1	100.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
MunchLife	4	1	25.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
TippyTipper	3	3	100.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
WorldClock	1	1	100.00%	1	100.00%	0	0.00%	0	0.00%	0	0.00%
TOTAL	**25**	**18**	**72.00%**	**6**	**24.00%**	**7**	**28.00%**	**3**	**12.00%**	**1**	**4.00%**

TABLE IV: Numbers and Percentages of Detected Hand-seeded Faults

Apps	# Faults	Android Mutation	% by Mutation	Monkey	% by Monkey	Dynodroid	% by Dynodroid	PUMA	% by PUMA	A^{3}E	% by A^{3}E
AlarmKlock	59	57	96.61%	31	52.54%	20	33.90%	26	44.07%	1	1.69%
AndroidomaticKeyer	25	17	68.00%	5	20.00%	16	64.00%	4	16.00%	1	4.00%
Jamendo	40	36	90.00%	7	17.50%	22	55.00%	22	55.00%	22	55.00%
JustSit	59	52	88.14%	16	27.12%	7	11.86%	26	44.07%	15	25.42%
Lolcat Builder	56	28	50.00%	3	5.36%	6	10.71%	19	33.93%	19	33.93%
MunchLife	52	41	78.85%	19	36.54%	22	42.31%	5	9.62%	3	5.77%
TippyTipper	47	47	100.00%	10	21.28%	11	23.40%	0	0.00%	0	0.00%
Tipster	64	51	79.69%	13	20.31%	22	34.38%	7	10.94%	12	18.75%
WorldClock	35	31	88.57%	26	74.29%	12	34.29%	12	34.29%	6	17.14%
TOTAL	**437**	**360**	**82.38%**	**130**	**29.75%**	**138**	**31.58%**	**121**	**27.69%**	**79**	**18.08%**

not. That is, the tests actually caused many of the mutants to produce incorrect images, but the test oracles were insufficient to see the images. This is not a problem with the technique, but a more general observability problem with test oracles. A similar observability problem with the test oracles resulted in failures caused by the tests that should have killed the mutant to not be observed. Li and Offutt [21] discussed this problem in the context of test oracles. In addition, our tests were not able to generate inter-app user events. For example, some faults can only be revealed when one app calls an Intent in another app, which our tests did not do. We also found an additional common fault across several subjects. Many apps have a settings and preferences menu to let users configure the apps. But sometimes the modified settings were not properly stored after the user changed them, leading to other incorrect behaviors of the app. We are trying to design additional mutation operators to encourage testers to design tests to ensure the settings menu works correctly.

VII. Related Work

This section describes relevant research in testing Android apps and mutation testing.

A. Testing Android Apps

Amalfitano et al. implemented MobiGUITAR [6], which models Android apps' GUI widgets as state machines and generates tests by applying graph testing techniques. Results showed that MobiGUITAR detected more faults than Android Monkey and Dynodroid. We were not able to run MobiGUITAR in our experimental environment, so could not use it.

Shahriar et al. [30] identified common memory leak patterns in Android apps, and designed memory leak guided fuzz testing. Results showed that their approach could effectively find memory leaks in Android apps.

Choudhary et al. [10] evaluated 7 automated Android test input generation tools on 68 subjects. Our evaluation has a different approach. We mined each subject's open source repository to collect naturally occurring faults. We also collected 437 crowdsourced faults from experienced Android developers. Due to the fact that we used a significantly larger set of benchmarks, with different natures of faults and different evaluation approaches, our results differ from the results in the paper of Choudhary et al.

B. Mutation Testing

Mutation testing has been studied, evaluated, and extended by many researchers for more than three decades. Ma et al. [22] designed mutation operators for Java by considering the characteristics of object-oriented programming (inheritance, polymorphism, and encapsulation), and implemented muJava, a mutation testing tool for Java. We used the muJava code base to develop our Android mutation testing tool.

XML is extensively used in web applications and Android apps to define layouts, store data, and configure the system. Lee and Offutt [20] applied mutation analysis to XML data and defined mutation operators specific to web component interaction, by mutating the interactions recorded in XML files. They designed test cases to detect the changes made to XML messages. The technique proposed by Lee and Offutt focused on checking the semantic correctness of the interactions between web components. Offutt and Xu [27] designed XML mutation operators that modify XML schemas to address the input data validation problem for web services. Unlike the approach of Offutt and Xu, we mutate XML files that define GUI layouts and configure the app, by considering them as a part of the source code.

VIII. Threats to Validity

Our evaluation has several threats to validity. We cannot guarantee that the subjects are representative. To ameliorate this, we chose apps that were diverse in size, functionality, and features. Also, all of them have been used as subjects by other researchers. Only one set of mutation-adequate tests was designed, so it is possible that the results of fault detection may

differ if using different tests. The strategy of using multiple test sets has been studied and called into question in other research [26]. Also, we cannot be sure that all the hand-seeded faults are representative faults. To avoid any possible bias, we recruited the freelancers through a very rigorous selection process, and eliminated all the hand-seeded faults that are exactly the same as mutants. We collected naturally occurring faults, inspected hand-seeded faults, and identified equivalent mutants, all by hand. Manual work could introduce mistakes that may affect the final results.

IX. Conclusion and Future Work

This paper evaluates the fault detection effectiveness of mutation testing for Android apps. We identified challenges in testing Android apps, analyzed common faults in Android app development, and defined novel Android mutation operators to help testers address these issues. We used two approaches to collect faults in Android apps for our experiment, mining open source repositories to collect naturally occurring faults, and crowdsourcing faults through a freelancer website. We share all of our faults online so that other Android testing researchers can benefit from these faults. We hope to keep improving this repository of Android faults and make it a community-owned benchmark to evaluate Android app testing techniques. The results of our experiment show that Android mutation testing is significantly effective at detecting faults. We suggest that testers should consider applying Android mutation testing in designing tests or in evaluating their existing tests if they value the quality of their apps.

Research into Android mutation testing is still continuing. We will keep improving the implementation of Android mutation operators, and designing additional novel mutation operators as discussed in Section VI. Additionally, we want to reduce the cost of Android mutation testing. One approach is to speed up the execution of mutation testing. Another is to eliminate redundant Android mutation operators and Java operators with empirical evaluation to reduce the number of test requirements (mutants).

Acknowledgment

We thank Freelancer.com and Robotium for helping and supporting our project. This work was partly funded by The Knowledge Foundation (KKS) through the project 20130085: Testing of Critical System Characteristics (TOCSYC).

References

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

[2] Android Monkey. https://developer.android.com/studio/test/monkey.html, last access May 2017.

[3] DAVdroid. https://github.com/bitfireAT/davdroid, last access July 2016.

[4] "Android apps on Google Play," 2016, http://www.appbrain.com/stats/number-of-android-apps, last access September 2016.

[5] "Metrics," 2016, http://metrics2.sourceforge.net, last access May 2017.

[6] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar – A tool for automated model-based testing of mobile apps," *Software, IEEE*, no. 99, pp. 1–1, 2014.

[7] P. Ammann and J. Offutt, *Introduction to software testing*, 2nd ed. Cambridge University Press, 2017, iSBN 978-1107172012.

[8] 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*. ACM, 2013, pp. 641–660.

[9] L. Bishop and D. Chait, "Fixing common Android lifecycle issues in games," 2015, https://developer.nvidia.com/fixing-common-android-lifecycle-issues-games, last access January 2015.

[10] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[11] 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.

[12] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of Android apps," in *Tenth Workshop on Mutation Analysis (Mutation 2015)*, April 2015, pp. 1–10.

[13] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing Android apps," *Information and Software Technology*, vol. 81, pp. 154–168, January 2017.

[14] M. Gómez, R. Rouvoy, M. Monperrus, and L. Seinturier, "A recommender system of buggy app checkers for app store moderators," in *Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems*, 2015, pp. 1–11.

[15] H. Gruber, "Android support lib bug causing crash on orientation change–A workaround," Online, February 2015, http://www.jayway.com/2015/02/03/android-support-lib-bug-causing-crash-orientation-change-workaround/, last access September 2015.

[16] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 389–398.

[17] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*. ACM, 2014, pp. 204–217.

[18] Kleiner Perkins Caufield & Byers, "Internet trends 2015," Online, May 2015, http://www.kpcb.com/internet-trends, last access September 2015.

[19] T. D. LaToza and A. van der Hoek, "Crowdsourcing in software engineering: Models, motivations, and challenges," *IEEE Software*, vol. 33, no. 1, pp. 74–80, Jan 2016.

[20] S. C. Lee and 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)*, Nov 2001, pp. 200–209.

[21] N. Li and J. Offutt, "Test oracle strategies for model-based testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, Apr. 2017.

[22] 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.

[23] 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*. ACM, 2013, pp. 224–234.

[24] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *Journal of Systems and Software*, vol. 126, pp. 57 – 84, 2017.

[25] R. Minelli and M. Lanza, "Software analytics for mobile applications–insights & lessons learned," in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp. 144–153.

[26] J. Offutt and M. E. Delamaro, "Assessing the influence of multiple test case selection on mutation experiments," in *Tenth IEEE Workshop on Mutation Analysis (Mutation 2014)*, Cleveland, OH, March 2014.

[27] J. Offutt and W. Xu, "Testing web services by XML perturbation," in *Proceedings of the 16th International Symposium on Software Reliability Engineering*, Chicago, IL, November 2005.

[28] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2011, pp. 5:1–5:6.

[29] D. Samudio. (2014) Automated Android Energy-Efficiency InspectiON. https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection, last access March 2017.

[30] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, Jan 2014, pp. 176–183.

[31] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST 2014)*, 2014, pp. 183–192.