MUTATION TESTING FOR ANDROID APPLICATIONS

by

Lin Deng
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

______________________________________  Dr. Jeff Offutt, Dissertation Director
______________________________________  Dr. Paul Ammann, Committee Member
______________________________________  Dr. Thomas LaToza, Committee Member
______________________________________  Dr. Ioulia Rytikova, Committee Member
______________________________________  Dr. Stephen Nash, Senior Associate Dean
______________________________________  Dr. Kenneth Ball, Dean, The Volgenau School
                                      of Engineering

Date: ________________________________  Fall Semester 2017
                                      George Mason University
                                      Fairfax, VA
Mutation Testing for Android Applications

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Lin Deng
Master of Science
Gannon University, 2011
Bachelor of Engineering
Renmin University of China, 2005

Director: Dr. Jeff Offutt, Professor
Department of Computer Science

Fall Semester 2017
George Mason University
Fairfax, VA
Dedication

I dedicate this dissertation to my wife Qing Guan, my parents Kunmei Deng and Yuqiu Su, my parents in law Kuiqi Guan and Chonghua Gao, my dear advisor Dr. Jeff Offutt and my other teachers and friends.
Acknowledgments

I would like to thank Dr. Jeff Offutt for advising me, supporting me, enlightening me to conduct world-class research in software engineering, showing me and teaching me how to be a great teacher, mentor, and leader. I want to thank Dr. Paul Ammann for giving me great help for my research and TA work throughout my entire Ph.D. study. I really enjoy working with you. I also thank my Ph.D. dissertation committee members Dr. Thomas LaToza and Dr. Ioulia Rytikova for giving me great suggestions for my proposal and dissertation, and spending your time on helping me reach every important milestone. I also would like to thank brothers and sisters in our research lab, Dr. Nan Li, Dr. Upsorn Praphamontripong, Dr. Nariman Mirzaei, and David Samudio. Discussing research ideas, finishing course works, writing research papers, conducting experiments, and chatting about any interesting topics with you will always be the unforgettable moments in my life. Especially, I would like to thank my wife, Qing Guan, and my parents, Kunmei Deng and Yuqiu Su. Completing a Ph.D. is a long, difficult, and bittersweet journey. Without your selfless love and support, I will never be able to finish it.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>Abstract</td>
<td>xii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Challenges in Testing Android Apps</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Problem Statement and Motivation</td>
<td>19</td>
</tr>
<tr>
<td>1.4 Hypothesis</td>
<td>21</td>
</tr>
<tr>
<td>1.5 Structure of This Ph.D. Dissertation</td>
<td>23</td>
</tr>
<tr>
<td>2 Background</td>
<td>25</td>
</tr>
<tr>
<td>2.1 Mutation Analysis</td>
<td>25</td>
</tr>
<tr>
<td>2.2 Android Applications</td>
<td>30</td>
</tr>
<tr>
<td>3 Related Work</td>
<td>32</td>
</tr>
<tr>
<td>3.1 Mutation Testing</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1 Application of Mutation Testing</td>
<td>32</td>
</tr>
<tr>
<td>3.1.2 Mutation Testing for eXtensible Markup Language (XML)</td>
<td>32</td>
</tr>
<tr>
<td>3.1.3 Reducing the High Cost of Mutation Testing</td>
<td>33</td>
</tr>
<tr>
<td>3.1.4 Minimal mutation analysis and dominator mutation score</td>
<td>34</td>
</tr>
<tr>
<td>3.2 Testing Android Applications</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Android Permissions and Security Issues</td>
<td>39</td>
</tr>
<tr>
<td>3.4 GUI Testing and Graphical Test Oracles</td>
<td>40</td>
</tr>
<tr>
<td>3.5 Mining Source Code Repositories and Bug Reports</td>
<td>42</td>
</tr>
<tr>
<td>3.6 Crowdsourcing in Software Engineering</td>
<td>43</td>
</tr>
<tr>
<td>4 Mutation Testing for Android Applications</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Mutating Android Applications</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Android Mutation Operators</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1 Event-based Mutation Operators</td>
<td>50</td>
</tr>
<tr>
<td>4.2.2 Component Lifecycle Mutation Operators</td>
<td>56</td>
</tr>
</tbody>
</table>
4.2.3 XML-related Mutation Operators ........................................... 59
4.2.4 Common Faults Mutation Operators ...................................... 64
4.2.5 Context-Aware Mutation Operator ........................................ 70
4.2.6 Energy-Related Mutation Operator ....................................... 72
4.2.7 Network-related Mutation Operator ..................................... 76
4.2.8 Summary ............................................................................. 77

5 Experiments .............................................................................. 79
5.1 Android Mutation Analysis Tool .............................................. 79
5.1.1 Functionality ....................................................................... 80
5.1.2 Architecture of muDroid ....................................................... 87
5.2 Empirical Evaluation of Android Mutation Testing ..................... 91
5.2.1 Empirical Subjects ............................................................... 92
5.2.2 Test Data Generation ........................................................... 96
5.2.3 Mutant Generation ............................................................... 97
5.2.4 Empirical Results and Discussion ........................................ 98
5.2.5 Threats to Validity .............................................................. 103
5.3 Experimental Evaluation of Fault Detection Effectiveness ............. 104
5.3.1 Experimental Subjects .......................................................... 106
5.3.2 Experimental Procedure ....................................................... 109
5.3.3 Collecting Naturally Occurring Faults ...................................... 110
5.3.4 Collecting Crowdsourced Faults ............................................ 111
5.3.5 Other Android App Testing Techniques .................................. 116
5.3.6 Experimental Results .......................................................... 117
5.3.7 Statistical Analysis ............................................................... 119
5.3.8 Analysis of Undetected Faults ............................................... 122
5.3.9 An Additional Common Fault .............................................. 126
5.3.10 Threats to Validity ............................................................. 127
5.4 Experimental Evaluation of Redundancy in Android Mutation Testing ........................................ 130
5.4.1 Experimental Subjects .......................................................... 131
5.4.2 Redundancy Scores ............................................................. 133
5.4.3 Experimental Procedure ....................................................... 134
5.4.4 Experiment Results and Discussion ...................................... 136
5.4.5 Re-evaluate the Effectiveness ............................................... 154
5.4.6 Threats to Validity ............................................................. 155

6 Conclusions and Future Work .................................................... 156
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Android Mutation Operators</td>
<td>49</td>
</tr>
<tr>
<td>4.2 IPR Default Values</td>
<td>52</td>
</tr>
<tr>
<td>5.1 Details of Empirical Subjects</td>
<td>94</td>
</tr>
<tr>
<td>5.2 Mutants Generated</td>
<td>97</td>
</tr>
<tr>
<td>5.3 Empirical Results</td>
<td>99</td>
</tr>
<tr>
<td>5.4 Empirical Results for Each Mutation Operator</td>
<td>101</td>
</tr>
<tr>
<td>5.5 An Example of FON Mutant</td>
<td>103</td>
</tr>
<tr>
<td>5.6 Details of Experimental Subjects</td>
<td>107</td>
</tr>
<tr>
<td>5.6 Details of Experimental Subjects</td>
<td>108</td>
</tr>
<tr>
<td>5.7 Numbers of Naturally Occurring Faults Collected for Each Subject App</td>
<td>112</td>
</tr>
<tr>
<td>5.8 Numbers of Hand-seeded Faults for Each App before Removing Mutants</td>
<td>113</td>
</tr>
<tr>
<td>5.9 Numbers of Hand-seeded Faults for Each App after Removing Mutants</td>
<td>114</td>
</tr>
<tr>
<td>5.10 Numbers and Percentages of Naturally Occurring Faults Detected by Android Mutation Testing</td>
<td>117</td>
</tr>
<tr>
<td>5.11 Numbers and Percentages of Crowdsourced Faults Detected by Android Mutation Testing</td>
<td>118</td>
</tr>
<tr>
<td>5.12 Numbers and Percentages of Naturally Occurring Faults Detected by Other Tools</td>
<td>120</td>
</tr>
<tr>
<td>5.13 Numbers and Percentages of Hand-seeded Faults Detected by Other Tools</td>
<td>121</td>
</tr>
<tr>
<td>5.14 Details of Experimental Subjects</td>
<td>132</td>
</tr>
<tr>
<td>5.15 Average Redundancy Scores</td>
<td>137</td>
</tr>
<tr>
<td>5.16 Average Redundancy Scores of Fail on Back (FOB)</td>
<td>139</td>
</tr>
<tr>
<td>5.17 Average Redundancy Scores of TextView Deletion (TVD)</td>
<td>140</td>
</tr>
<tr>
<td>5.18 Average Redundancy Scores of Orientation Lock (ORL)</td>
<td>142</td>
</tr>
<tr>
<td>5.19 Average Redundancy Scores of Activity Lifecycle Method Deletion (MDL)</td>
<td>143</td>
</tr>
<tr>
<td>5.20 Average Redundancy Scores of Unary Arithmetic Operator Deletion (AODU)</td>
<td>146</td>
</tr>
<tr>
<td>5.21 An Example AODU Mutant</td>
<td>148</td>
</tr>
</tbody>
</table>
5.22 Average Redundancy Scores of Button Widget Deletion (BWD) and Button Widget Switch (BWS) ................................................. 149
5.23 Average Redundancy Scores of Constant Deletion (CDL), Conditional Operator Deletion (COD), Operator Deletion (ODL), and Variable Deletion (VDL) .......................................................... 150
5.24 Example ODL Mutants .......................................................... 151
5.25 An Example CDL and VDL Mutant ......................................... 151
5.26 An Example COD Mutant ....................................................... 151
5.27 Average Redundancy Scores of Unary Arithmetic Operator Insertion (AOIU) and Logical Operator Insertion (LOI) .......................... 152
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An Example Activity</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>An Example Broadcast Receiver</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>An Example Content Provider</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>An Example Failure of Inappropriately Handling Activity Lifecycle</td>
<td>8</td>
</tr>
<tr>
<td>1.5</td>
<td>Lifecycle of Activity in Android apps</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>Lifecycle of Service in Android Apps</td>
<td>10</td>
</tr>
<tr>
<td>1.7</td>
<td>XML Layout File</td>
<td>11</td>
</tr>
<tr>
<td>1.8</td>
<td>An Example Context-aware Input</td>
<td>13</td>
</tr>
<tr>
<td>1.9</td>
<td>An Example of Android Apps Adapting to Orientation Change</td>
<td>14</td>
</tr>
<tr>
<td>1.10</td>
<td>Android App Requests Permissions</td>
<td>14</td>
</tr>
<tr>
<td>1.11</td>
<td>An Example of Installing Android Apps with Permissions</td>
<td>15</td>
</tr>
<tr>
<td>1.12</td>
<td>Three Android System Buttons</td>
<td>16</td>
</tr>
<tr>
<td>1.13</td>
<td>Discussions on the Issue of Crashes after Back Button Is Clicked</td>
<td>17</td>
</tr>
<tr>
<td>1.14</td>
<td>An Example Energy Bug</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>General Mutation Testing Process</td>
<td>26</td>
</tr>
<tr>
<td>2.2</td>
<td>Relational Operator Replacement Example</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>An Example of Killing a Mutant</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Performing Mutation Analysis on Android Apps</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Intent Payload Replacement Mutation Operator</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>Intent Target Replacement Mutation Operator</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>OnClick Event Replacement Mutation Operator</td>
<td>54</td>
</tr>
<tr>
<td>4.5</td>
<td>OnTouch Event Replacement Mutation Operator</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>An Example MDL Mutant</td>
<td>57</td>
</tr>
<tr>
<td>4.7</td>
<td>An Example SMDL Mutant</td>
<td>58</td>
</tr>
<tr>
<td>4.8</td>
<td>An Example BWD Mutant</td>
<td>60</td>
</tr>
<tr>
<td>4.9</td>
<td>Button Widget Deletion (BWD) and EditText Widget Deletion (TWD) Example</td>
<td>61</td>
</tr>
</tbody>
</table>
Along with the significantly widespread of Android devices, Android applications (apps) also dominate the global market, in terms of the users, developers, app releases, and downloads. However, the quality of Android apps is a growing and significant problem. Many apps are released to the market with severe software faults that result in crashes, incorrect behaviors, and security vulnerabilities. Testing Android apps differs from testing traditional software programs because Android apps include new programming features and structures never used before. New types of software faults may be introduced into Android apps by these unique characteristics, but existing software testing techniques and simple testing coverage criteria cannot detect these new software faults, or help developers deliver high quality Android apps.

This research investigated the new development frameworks, unique programming features, common programming faults, and novel characteristics of Android apps, and designed Android mutation testing, a more sophisticated testing strategy for Android apps than current practice. An Android mutation analysis tool, muDroid, was designed and implemented, which includes 17 novel Android mutation operators and extends 19 Java traditional method-level mutation operators.

Using three experimental studies, this research shows that Android mutation testing ca-
n design test cases that are very effective at detecting both naturally occurring faults and crowdsourced faults for Android apps. Also, Android mutation testing can provide an effective evaluation criterion for assessing other Android apps testing techniques. Redundant or ineffective Android test cases can be filtered out with Android mutation testing, and ultimately, our ability to deliver high quality Android apps can be improved.
Chapter 1: Introduction

1.1 Introduction

Mobile applications (mobile apps) are software programs that are installed and executed on mobile devices, such as smartphones, tablets, smart TVs, and smart watches. Because the prices of mobile devices are gradually becoming cheaper, and the mobile networks are rapidly expanding to almost every country, more people around the world are acquiring mobile devices. Sometimes, one person may possess multiple mobile devices: he or she may use a smartphone for calling family members and friends, may have a smart TV for watching news and movies, and may wear a smart watch for tracking movements and physical activities.

Cisco recently reported that, in 2016, the total number of global mobile devices reached 8.0 billion, exceeding the world population (7.4 billion) in the same year [56]. Not surprisingly, people have already started to use mobile devices more frequently than desktops and laptops. Since 2008, KPCB, an American venture capital firm, has tracked the time that American people spend on mobile devices, desktops, and laptops. Its report shows that in 2015, on average, US adults spend 2.8 hours every day on their mobile devices, compared to only 0.4 hours daily in 2010, and 2.4 hours devoted to PCs in both 2010 and 2015 [92].

Mobile devices have evolved from slow devices with a tiny, low-resolution, and black and white screens that are only capable of voice calling, texting, and browsing news without any pictures or animations, to powerful portable computers and entertainment terminals. Mobile devices are improving people’s life, including the way we shop, entertain, travel, communicate, and make friends. They are becoming critical to humans’ daily lives. A survey conducted by TechRepublic in 2016 shows that 94% of the respondents use mobile devices in their work [148]. Another survey from Boston Consulting Group (BCG) demonstrates
that people are increasingly unwilling to live without mobile devices, even at the expense of losing traditional needs. More than 55% of the participants would forgo dining out for 12 months rather than lose their smartphones, and 45% would put off going on vacation. More than three in 10 would stop seeing their friends in person, and 46% would give up a day off per week [1].

Android, iOS, and Windows Mobile are three major operating systems designed for mobile devices. According to International Data Corporation (IDC), Android devices dominate the global market, due to the diversity of brands and devices, and a wider range of prices compared to other products. In the third quarter of 2016, the worldwide unit shipments of Android devices reached 86.8% of the market, while iOS and Windows Mobile only had 12.5% and 0.3% [84].

Android apps are software programs that execute on the Android operating system. Due to the popularity of Android devices, Android apps also grew tremendously, in terms of the numbers of the users, developers, and apps. In September 2015, Google reported that more than one billion users had used Google Play store within 30 days [49, 77, 126]. Evans Data Corporation (EDC) reported that the total number of software developers developing Android apps reached 5.9 million in 2016 [70]. In March 2017, the total number of Android apps on the Google Play store exceeds 2.8 million [4].

Due to the exceptional convenience of mobile devices, people can use mobile apps at anytime, day and night, from anywhere, whether sitting on a train or flying on a plane. The increasing popularity of mobile devices, greatly enhanced hardware specification, and the new generation of mobile networks, inevitably attracts IT companies to invest in developing mobile apps for earning more profits from mobile users.

However, due to the sheer volume of Android apps, quality becomes a growing and significant problem. Research of Gómez et al. reveals that many Android apps released in the Google Play Store contain severe software faults, resulting in failures during use, including runtime crashes, incorrect behaviors, and security vulnerabilities [76]. Similar to other types of software, the inferior quality results in numerous undesirable outcomes. 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 appears to be higher. In 2014, 20% of apps were launched only once before uninstalled, and 61% were used less than ten times [112]. Considering the huge amount of apps available online, the total number of apps discarded by users is tremendous. Even though a user can delete an app for all kinds of reasons, such as poor usability and lack of expected features, software failure is definitely an essential factor that makes users decide to remove the app from their mobile devices.

Usually, if an app keeps crashing on users’ devices, most users would decide to uninstall it. Gradually, it will disappear from the market. From the perspective of software reliability, any downtime is a threat to enterprise revenue. The apps of eBay introduced $28 billion in total sales in the year of 2014 [142], which means that, in one year, even 0.1% of apps downtime will incur a loss of $28 million. For another example, mobile banking apps have become increasingly popular year by year. Transferring balance, depositing checks, and paying bills can all be done with a couple of seconds on apps. However, their quality is also critical to everyone. In the year of 2013, the mobile app of Royal Bank of Scotland had 34 times of massive failure events, which affected millions of its customers [53, 155]. There was no report about the financial loss of the bank. It is difficult to quantify the cost of the failures, but the costs are certainly high. Thus, quality of mobile apps is critical to developers, vendors, and customers.

1.2 Challenges in Testing Android Apps

Most Android apps are written in Java, but Android apps have unique characteristics and additional features beyond traditional software, including the way they are developed, tested, distributed, and installed [122]. In particular, these new characteristics and features make the entire testing process for Android apps different, and bring several challenges to developers and testers when testing Android apps. This section introduces the unique characteristics and associated challenges identified during the research into Android apps,
which need to be appropriately addressed when developing and testing Android apps.

1. **Four types of components**

Android apps contain four types of components: *Activities, Services, Broadcast Receivers,* and *Content Providers.* For example, in the manifest file of an app, the app designates an *Activity* to work as the “main class” along with security permissions and subscriptions to intent broadcasts. These components are written in Java, and provided by the Android Software Development Kit (SDK). Developers can extend these super classes (components) and provide their own implementation based on software requirements.

An *Activity* presents a screen, Graphical User Interface (GUI), to the user based on one or more layout designs. The GUI may include widgets such as buttons, textviews, and other advanced artifacts. Figure 1.1 shows an example *Activity.* Developers populate the GUI by overriding the *Activity* lifecycle methods, and can add programming logic for the GUI. Additionally, developers separate the visual structure from the behavior by using eXtensible Markup Language (XML) to declare the app’s layout.

*Services* do not have a GUI, and run in the background. They do not interact with the screen. Thus, they are usually used to perform long-term running tasks, such as playing music or triggering alarm clocks. A *Service* can be started by other components, including *Activities* and *Broadcast Receivers.* *Services* also have a different lifecycle from an *Activity,* which is addressed in Section 2.

A *Broadcast Receiver* is used to subscribe the app to *Intents* broadcast by the Android system or other apps, such as low battery. Figure 1.2 gives an example *Broadcast Receiver.*

A *Content Provider* supplies and manages structured data for other apps. These data are stored in file systems or databases, including calendars, photographs, contacts, and stored music. For example, Figure 1.3 shows an example of using *Content Provider* to access files stored in the device. The figure on the left is a Facebook screenshot [17].
When the user clicks the *Photo* button, the *Content Provider* is called and provides all the photographs saved in the device. Then, the user can pick one and upload it to Facebook.

Some research, such as MobiGUITAR [41], considers Android apps the same as GUI software, and applies GUI testing techniques to test Android apps. These approaches might be suitable for testing *Activities*, as it is displayed on a screen. However, they cannot test other components comprehensively.

2. **Unique lifecycles of Android components**
   
   Unlike other types of software, the Android operating system requires major components of Android apps to behave according to a pre-defined lifecycle [6]. If a component’s lifecycle is not appropriately handled, it is very likely to cause unexpected
issues when users swap between different apps, pause then resume an app, or turn on
the mobile device from sleep mode [51]. Particularly, the flow of continuity in some
Android apps is critical to users, such as games. For example, Figure 1.4 illustrates
a gaming app that incorrectly handles the lifecycle of Activity. The picture on the
left shows the game progress, where the user is going to win the card game. However,
when the user finishes the call and returns to the game, the app is not able to pro-
perly handle the lifecycle by calling the correct methods to restore the game progress
from where it is interrupted. Instead, it restarts from the beginning, as shown in the
picture on the right.

Figure 1.5 shows the lifecycle of an Activity in the form of states and event transitions.
An Activity has three states connected by different conditions: Running, Paused, and
Stopped. After an Activity is launched, three methods, onCreate(), onStart(), and
onResume(), need to be sequentially called before the Running state is reached, i.e.,
the user can see the Activity on the screen. The onPause() method sends the Activity to the Paused state, where it can return to the Running state with onResume(), or goes to the Stopped state with onStop(). When the user wants to switch back to the stopped Activity, three methods, onRestart(), onStart(), and onResume(), need to be sequentially called to bring the Activity back to the screen. Or, when the Activity will not be used anymore, it can exit with an onDestroy() method.

Unlike an Activity, which displays a screen to the user, Services are invisible and perform long-term running tasks in the background, such as playing music. Services also behave according to pre-defined lifecycle, but it is not the same as the lifecycle of Activity. Figure 1.6 shows the lifecycle of two different types of Services as a collection of event methods and states.
Services come in two types: unbounded and bounded. An unbounded Service is launched after the Android system executes the onCreate() and onStartCommand() methods of the Service, 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 the Service. 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.

In addition, the two forms of Services are not mutually exclusive. A bounded Service can also be started with the `onStartCommand()` method. Then, a client can bind to the Service again after the `onUnbind()` method is called as long as the Service is still active. The Android system will call either the `onRebind()` or the `onBind()` method to rebind the Service. However, if this Service is no longer needed, it needs to either stop itself or be stopped by its client, but not by the Android system.

Since the communication between Service and other components is critical to the smooth execution of Android apps, particularly for apps that require the correct running of Services, such as music players, email clients, and alarm clocks, developers
must appropriately implement *Services* according to the *Service* lifecycles. The study into an alarm clock app identified software faults in its *Services* that caused the app to sound at a wrong time, and some leading to runtime crashes.

3. **Intensive use of eXtensible Markup Language (XML) files**
   
   Even though most Android apps’ source code is written in Java, XML files are also intensively employed by Android apps for program configuration, user interface (layout) specification, and temporary data storage. For example, Figure 1.7 shows an example XML layout file in an Android app. This XML file defines the general layout structure of current *Activity* as *RelativeLayout*, then creates a *TableLayout* inside it. Several *Button* and *TextView* widgets with different sizes and fonts are also defined in the *Activity*.

   Depending on the design of different projects, it is possible for an Android app to include even more XML files than Java source files. However, using XML files for these purposes is relatively new. For example, programs with Graphical User Interfaces
Figure 1.7: XML Layout File
(GUIs) implemented with Java or C# rarely use or include XML files. Consequently, no testing techniques designed for testing traditional Java programs consider source code other than Java within the same project. Additionally, there is no test coverage criterion to measure the coverage information for XML files used in Android apps. For a given test, we can easily observe whether a statement or a logical branch is executed or not. However, for an XML file in the same project, we do not have techniques to evaluate coverage of it. Indeed, not testing XML files of Android apps may result in unexpected failures.

4. **Context-aware characteristics**

Another important distinct characteristic of Android apps is that they are context-aware. Apps receive a variety of input data from its physical environment through different sensors, such as the linear acceleration, Global Positioning System (GPS) location, and rotation. For example, context-aware apps behave differently when the phone is moving in a vehicle versus sitting at a desk. Figure 1.8 shows another real example with a widely used Android app, Yelp [36]. Given the exactly same test input, selecting *Restaurants*, choosing *Current Location*, then *Search*, standing at different locations leads to entirely different results. Undoubtedly, the location input received from the device directly impacts the results. And these types of data are not fed by users, rather from sensors in the device itself. Moreover, this difference in behavior is not reflected directly in the app’s source code; rather the difference is in how often the app receives an event notification. In a sense, 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.

5. **Two types of screen orientation**

The ability to change orientation is a key distinct characteristic of mobile devices. Almost every device has two types of screen orientation, landscape and portrait. Screen orientation is switched automatically when a device detects a change in the way it is held by the user, unless it is manually locked by the user. Many Android apps are
Figure 1.8: An Example Context-aware Input
designed to have different user interfaces to adapt to the orientation change event. For example, Figure 1.9 shows that a simple calculator (left) with portrait orientation becomes a scientific calculator (right) with landscape orientation. Unlike traditional software, which does not take orientation into account, testing Android apps must consider this unique feature of orientation changing. This is because it is highly likely to cause different types of failures, such as immediately crashing after switching the orientation [78].  

6. Android apps’ permissions

For protecting Android operating system’s security and users’ privacy, every Android app is executed in a separate sandbox with limited permissions. If an app needs to
access system resources or user data, it must explicitly declare the requested permissions in its AndroidManifest.xml file. For example, Figure 1.10 shows a part of an app’s AndroidManifest.xml file that requests three permissions: \textit{WAKE	extunderscore LOCK}, \textit{MODIFY	extunderscore AUDIO	extunderscore SETTINGS}, and \textit{VIBRATE}. Usually, when an app is installed on an Android device, the Android system will ask the user to decide whether to grant the permissions.

\begin{verbatim}
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
    <uses-permission android:name="android.permission.VIBRATE" />
</manifest>
\end{verbatim}

Figure 1.10: Android App Requests Permissions

The screenshot on the left of Figure 1.11 lists the permissions requested by Facebook [17], including Calendar, Contacts, Location, Files, Camera, etc. As the top Android
app for social networking, it is reasonable for Facebook to access these system resources and user data. However, the screenshot on the right of Figure 1.11 shows that UNO [34] wants to access users’ Identify, Location, and Files, which are not necessary to a card game.

![Figure 1.11: An Example of Installing Android Apps with Permissions](image)

This mechanism, however, usually doesn’t ensure Android’s security as expected. Many users simply accept every request. Undoubtedly, this mechanism results in several testing issues and security vulnerabilities that can be used to exploit Android devices. Johnson et al. conducted an analysis of 141,372 Android apps, and concluded that the majority of apps do not come with an appropriate set of permissions. More than 54% of the apps analyzed requested extra permissions that were unnecessary to the execution [86]. The permission mechanism is supposed to deliver security to users, whereas some developers take advantage of it to perform malicious activities, which
are not usually found by traditional testing techniques.

7. Event-based programs, Intent, and Back button

Android apps can be considered as event-based programs, because their execution flows rely heavily on events initiated by different user actions, such as clicking, tapping, and dragging. Event handlers are implemented to react to these user events, so improperly designed event handlers could lead to incorrect software behaviors. Moreover, Android apps use Intent objects to facilitate the communication between components, e.g., initiating another Activity from a component. Usually, an Intent object carries the data demanded by the target component. It is evident that Intent objects play a critical role in Android apps, and need to be tested effectively.

In addition, as shown in Figure 1.12, every Android device is equipped with three system buttons: Back (left), Home (middle), and Recents (right).

![Three Android System Buttons](image)

Figure 1.12: Three Android System Buttons

The Home button lets users return to the main home screen. The Recents button displays a scrollable screen containing recently suspended or closed apps in reverse chronological order. Users can switch between different apps by tapping on them, or terminate a suspended app by swiping it away. The Back button enables users to move to previous screens, similar to the back button on web browsers. A particular stack of Activities managed by the Android operating system facilitates this Back action. Because the Back button interrupts the usual execution flow and is usually not on a happy path (the default scenario that should happen under normal use without exceptions) from the perspective of software design, many testers might overlook its impact. A common Android fault is for an app to crash when the Back button
is clicked. Figure 1.13 shows that on StackOverflow [31], a leading website where professional and enthusiast programmers all over the world discuss technical issues and look for solutions, there are nearly 500 discussion threads about how to solve the crash after the Back button is clicked in Android apps.

Figure 1.13: Discussions on the Issue of Crashes after Back Button Is Clicked

8. **Varied Network Connections**

Usually, an Android smartphone is equipped with several forms of network connections, most commonly cellular data and WiFi. By default, whenever a WiFi connection is available, the Android system will first attempt to transmit data through WiFi, as WiFi connections use relatively less energy, work at a higher download and
upload speeds, and cost less. If WiFi is not available, unless the cellular data is
disabled manually, the Android system will automatically switch to a cellular data
connection, which has a high network speed, is more expensive, and uses more energy.
This switching may cause problems in different scenarios. For example, Internet Pro-
ocol (IP) addresses are different when the device is connected to different networks.
It is very common that the network a smartphone switches to has difficulty resuming
the connection and fails to continue the previous unfinished tasks. Interrupted in-
stallations, leading to partial and unstable installations, and incomplete or duplicate
mobile shopping orders, are very common. Many developers and testers overlook this
situation, which results in the fact that users have to remember to stay connected to
one network without moving or switching before an important app finishes its tasks.

9. **Limited battery life**

Unlike PCs, which have a constant power supply, mobile devices have to rely on limited
battery power. No execution can happen when the battery is out of power. Therefore,
developers have to take battery usage into account when implementing their apps.
However, many developers overlook the battery usage. Several researchers found that
certain inappropriate programming practices could increase the energy consumption
of Android apps [107]. Some research projects label them as *energy bugs* [48,139,153].
Even though energy bugs do not downgrade the functionality of an app, they can
severely impact the Android system, and shorten the availability of the entire system.
Figure 4.19 compares two energy consumption diagrams captured from the same app.
Initially, the app has an energy bug (top), so that after it transits from an active
state to an idle state, it does not release system resources it requested, and keeps
consuming battery energy. Ideally, an app is expected to obtain system resources
when active, and release them when idle. These states differ from background and
foreground states of Android components because an app can be active while also in
the background, e.g., *Services*. The diagram at the bottom of Figure 4.19 shows the
expected energy consumption after fixing the energy bug.
1.3 Problem Statement and Motivation

Android apps employ new programming features that have never been used by traditional software before. These unique characteristics of Android apps lead to new types of faults, which are usually not revealed by existing software testing techniques or simple testing coverage criteria. Consequently, software testing techniques targeting traditional software are not sufficient to test Android apps, and we do not have sufficient knowledge and experience for how to thoroughly test them. This results in weak and ineffective testing.

Recently, the software engineering community has shown significant interest in inventing software testing approaches specific for Android apps. Some techniques have been proposed...
to address one or several challenges described in section 1.2, such as energy bugs and the event-driven nature of Android apps. However, there is still a lack of effective and usable techniques to evaluate their proposed test selection strategies, or to ensure a reasonable number of effective tests.

Problem Statement:
Currently, traditional software testing techniques are insufficient for testing Android apps due to the novel characteristics of Android apps.

Random value generation is still state-of-the-art in testing Android apps. Android Monkey [8], a random event generation tool released by Google, is one of the most widely used methods to test Android apps and evaluate new approaches, in both the industry and research communities [39, 40, 83]. However, the effectiveness of random testing is always questionable. Additionally, as the test inputs are randomly generated, then sent to the app under test, it is almost impossible to set and check any expected values with random testing. Therefore, random testing can usually only detect crashes and exceptions. However, Li and Offutt found that only about 30% failures are runtime crashes [105].

Model-based approaches have been extended to test Android apps as well, such as using state machines [147] and GUI models [41, 46, 160]. However, one precondition of model-based approaches is the availability of models. If the models are not available or insufficient, it is very hard to carry out model-based testing. In addition, it is very likely that different people would abstract different models for the same app.

Some papers extend symbolic execution into testing Android apps. Mirzaei et al. [124] created stubs and mock classes to make Android apps run on Java PathFinder (JPF) [20]. Merwe et al. [151, 152] developed JPF-Android by extending JPF to verify Android apps. But the state explosion problem limited its ability to generate complex tests inputs. Details about related work are discussed in Section 3.2. Right now, few of the proposed approaches are actively maintained or used by industry in actual Android apps testing.

The motivations of this research are:
1. To develop a new approach to testing Android apps by investigating the programming framework, unique features, and novel characteristics of Android apps

2. To provide more sophisticated testing than current practice

3. To supply an evaluation criterion for assessing other Android app testing techniques

4. To filter redundant and inefficient Android test cases

5. To ultimately improve our ability to deliver quality Android apps through stronger testing

1.4 Hypothesis

This research investigates the following hypothesis:

Research Hypothesis:
Mutation testing of Android apps can reveal more faults than existing testing techniques can.

In other software domains, mutation analysis has been found to excel at designing test cases, as well as evaluating other testing techniques. For example, the empirical results of Mathur and Wong [121, 158] showed that all the 120 randomly generated mutation adequate test sets satisfied all-uses criterion, while none of the 120 randomly generated all-uses adequate test sets was mutation adequate. Also, their empirical study found that mutation testing detected more faults than all-uses testing criterion did. Similarly, Offutt et al. [129] found that mutation adequate test sets detected 16% more faults than all-use test sets did.

If the hypothesis is true, mutation testing could guide developers and testers to design more effective tests than before, and to improve the quality of Android apps. Also, Android mutation coverage could be used as a testing criterion to evaluate and trim redundant tests generated by other testing techniques.
This research conducted three experimental evaluations to verify the hypothesis and to investigate Android mutation testing from different perspectives. Chapter 5 provided the detail of each experimental study.

The first experimental study investigated the feasibility of applying mutation analysis to testing Android apps. Particularly, it focused on verifying whether Android mutation testing can be used to evaluate test cases designed with other testing criteria. It addressed the following research questions:

- **RQ1**: Is it feasible to test real-world Android apps with mutation analysis?
- **RQ2**: How effective can test cases designed with traditional testing criteria be at killing mutants generated by Android mutation testing?

After exploring the applicability of Android mutation testing, the second experimental evaluation investigated the effectiveness of Android mutation testing. Specifically, the second evaluation includes 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. In addition, to make the studies more comprehensive, this evaluation used a combination of naturally occurring faults and crowdsourced faults created by experienced Android developers. This evaluation answers the following research questions:

- **RQ3**: How effective is Android mutation analysis at testing Android apps? Specifically, how many faults can be detected by mutation-generated tests?
- **RQ4**: 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?
- **RQ5**: Is there any difference between using naturally occurring faults and using crowdsourced faults in empirical evaluations?

After evaluating the effectiveness of Android mutation testing, the third evaluation investigated the possibility of reducing the high cost of Android mutation testing by searching
for and excluding redundant mutation operators. “Cost” in mutation analysis can be in different forms. Chapter 3 discusses the cost of mutation testing and related research work on reducing the cost. This study considers the computational time and effort as the major cost of Android mutation testing for executing tests against mutants. This evaluation answers the following research questions:

- **RQ6:** How many mutants of one particular type can be killed by tests created to kill another type of mutants?
- **RQ7:** Which types of mutants are less likely to be killed by tests created to kill other types of mutants?
- **RQ8:** Are any Android mutation operators redundant enough to be excluded, or can any be improved? In particular, can the mutants generated from this mutation operator always be killed by tests created to kill another type of mutant?

### 1.5 Structure of This Ph.D. Dissertation

The rest of this dissertation is organized as follows. Chapter 2 provides background on mutation analysis and Android apps. Mutation analysis is the key foundation of this research, and the major idea of this research is built upon the mutation analysis. Android apps are the research subjects and the research domain of this dissertation. Chapter 3 discusses the related research work in mutation testing, testing Android apps, Android permissions and security issues, testing GUI software, mining source code repositories, and crowdsourcing in software engineering. Chapter 4 introduces the approach of Android mutation testing, presents the design of novel Android mutation operators, and describes the system constructed for Android mutation testing. Chapter 5 presents a set of empirical studies of Android mutation testing. The first evaluation explores the feasibility of Android mutation testing. The second study evaluates the fault detection effectiveness of Android mutation testing and other four existing Android testing techniques with two types of faults: naturally occurring faults and crowdsourced faults. The third study investigates the redundant
mutation operators in Android mutation testing. Finally, Chapter 8 concludes this dissertation with a summary of the major contributions of this research, and finishes by suggesting future research work.
Chapter 2: Background

This research applies an existing technique, mutation testing, to a new type of software, Android mobile apps, to design effective tests. This chapter presents background on mutation testing and Android apps.

2.1 Mutation Analysis

Software testing is a set of activities that are used to validate and verify that a software system is developed in accordance with its requirements. In the field of software testing, mutation testing, first proposed in the paper by DeMillo et al. in 1978 [64], is one of the most effective testing techniques. It is a syntax-based testing technique, that has been empirically found to be exceptionally effective at helping testers generate high-quality tests, and at evaluating pre-existing tests designed by other testing techniques [43].

Mutation testing modifies a software artifact such as a program, requirement specification, or a configuration file to create new versions, called mutants. One single change made in the software artifact creates a first-order mutant, whereas multiple changes create a higher-order mutant. This research only focuses on first-order mutation testing.

Figure 2.1 illustrates a general mutation testing process. First, the original program under test $P$ is modified to create mutants, denoted as $P'$ in Figure 2.1. The mutants $P'$ are usually intended to be faulty versions and are created by applying rules that specify how the changes can be made on the software artifact. These rules are called mutation operators. For example, Figure 2.2 shows an example Relational Operator Replacement (ROR) mutant for Java. ROR replaces each instance of a relational operator (for example, $<$) with all other relational operators ($\leq$, $\geq$, $>$, $!=$) plus trueOp and falseOp, which set the condition to true and false [43]. Mutation operators sometimes create changes that mimic
typical programming mistakes, and sometimes introduce changes that encourage common
test heuristics by challenging testers to design test inputs that are likely to find faults. Well
designed mutation operators can lead to very powerful test cases.

Then, testers design and execute test cases $T$ against both the original program $P$ and
the mutated versions $P'$. If a test $t$ in the test set $T$ causes different outputs on the original
$P$ and a mutant $p$ in $P'$, the test $t$ is said to kill $p$. Figure 2.3 provides a program example
about designing a test to kill a mutant. This program looks for the last index of zero in
the array $x$. ROR replaces the condition statement $i \geqslant 0$ in the for-loop with $i > 0$,
which makes the program fail to check whether the first element in the array $x$ is zero. The
ineffective test in Figure 2.3 defines the array $x$ to be [1, 1, 2], which results in the same
output (-1) for the original program and the mutant. Obviously, even though the mutant
skips the first element in the array $x$, this ineffective test is not able to reveal the fault. The
second test assigns the array $x$ to be [0, 1, 2], which kills the mutant as the original output
is 0, while the mutant output is -1. This killing test is very effective because it helps the
tester to explore the boundary case.

Specifically, in this research a test $t$ must satisfy four conditions to kill a mutant, which
is called the RIPR model [43,105]:

Figure 2.1: General Mutation Testing Process
public int Min (int A, int B)  
{  
    int minVal = A;  
    if (B > A) // ROR mutant  
        minVal = B;  
    return minVal;  
}  

Original

public int Min (int A, int B)  
{  
    int minVal = A;  
    if (B < A)  
        minVal = B;  
    return minVal;  
}  

Mutant

Figure 2.2: Relational Operator Replacement Example

1. **Reachability**: the test must reach the location where the mutant locates

2. **Infection**: after executing the location of the mutant, the state of the program must be incorrect

3. **Propagation**: the infected state must cause some output or final state of the program to be incorrect

4. **Revealed**: the test must observe at least part of the incorrect portion of the final program state

Particularly, in this research, the **Revealed** condition is required to kill a mutant.

The more mutants a test set can kill, the more effective the test set is. Some mutants always produce the same output as the original program, so that they cannot be killed by any tests. These mutants are called **equivalent mutants**. The percentage of non-equivalent mutants killed by the test cases $T$ is called the **mutation score**. The mutation score ranges from 0% to 100%, which is a direct quantitative indicator about the effectiveness of a test
int lastZero (int [] x)
{
    for (int i = x.length-1; i >= 0; i–)
    {
        if (x [i] == 0)
            return i;
    }
    return -1;
}

int lastZero (int [] x)
{
    for (int i = x.length-1; i > 0; i–) // ROR mutant
    {
        if (x [i] == 0)
            return i;
    }
    return -1;
}

Input: x = [1, 1, 2]
Original output: -1
Mutant output: -1

Ineffective Test

Input: x = [0, 1, 2]
Original output: 0
Mutant output: -1

Killing Test

Figure 2.3: An Example of Killing a Mutant

set. From another perspective, mutants can be considered as test requirements that need to be covered or satisfied (killed, in mutation testing) by the test cases designed by developers and testers.

Some mutants behave exactly the same as the original program on every possible input so that no test cases can kill them. 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* mutants can usually be avoided if the mutation operators are well designed and appropriately implemented, some do occur. A mutation system must be prepared to recognize stillborn mutants and remove them from consideration. Some tests cannot kill any mutants, or can only kill mutants that have been already killed by other tests in the same test set $T$. These tests are called *ineffective*. Equivalent mutants and ineffective tests should be eliminated.

Next, if any mutants are still alive, the tester needs to design additional test cases to kill them. Otherwise, if all the mutants $P'$ are dead, the test set $T$ is called *mutation adequate*, i.e., the mutation adequacy score is 100%. The mutation adequate test set $T$ can be further used to verify the original program $P$ and identify any possible faults.

Mutation testing has been measured to be stronger than other test criteria. One source of that strength is that it does more than just apply local requirements, such as reaching a statement or touring 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*) [43, 65, 125].

Mutation operators have been created for many different languages, including C, Java, and Fortran [38, 89, 91, 113]. More details of related work about mutation testing are discussed in section 3.1. In this research, new mutation operators for Android apps are designed by investigating the novel features and unique characteristics of Android apps.
2.2 Android Applications

The Android operating system is developed based on the Linux kernel, and includes middleware, system libraries, Application Programming Interfaces (APIs), and several pre-installed applications. It is now the leading mobile operating system installed on smartphones, tablets, smart TVs, wearable devices, and automobiles [3]. The latest version of the Android operating system is 7.0 (Nougat). Android apps are commonly written in Java, and compiled into bytecode that can execute on the Java Virtual Machine (VM). There are two versions of Java Virtual Machines used by the Android operating system. Before version 4.4 (KitKat), Android used the Dalvik Virtual Machine. After that, version 5.0 (Lollipop) uses Android Runtime (ART) to replace Dalvik [13]. However, Google confirms that most apps developed for Dalvik should work without any changes under ART [13]. 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 provides developers a rich Software Development Kit (SDK) with a set of tools to compile Java based source code, resource files (e.g., pictures, audios, and videos), and data, and install on Android devices. 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 calendars, 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 calls go through the Android messaging service, rather than being explicitly present in the app. 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 relevant information about the app, including permissions, configurations, and descriptions of the apps’ components.
Chapter 3: Related Work

3.1 Mutation Testing

Mutation testing is a syntax-based testing technique, and has been studied, evaluated, and extended by many researchers for more than three decades. This section introduces papers in mutation testing that are related to Android mutation testing.

3.1.1 Application of Mutation Testing

Mutation testing has been applied to many programming languages, including Fortran 77 [65,91], C [62], Java [90,114], JavaScript [123], AspectJ [103], and web applications [141]. Several papers also extend mutation analysis to models, such as Finite State Machines [71,82], statecharts [149], Petri nets [72], timed automata [128], and Aspect-oriented models [108]. A new application of mutation testing is to fix software faults automatically [101]. Beyond the domain of software, mutation testing also has been applied to access control policies [120] and spreadsheets [37]. In addition, Oliveira et al. designed a specific set of mutation operators for GUI-based applications [136]. However, this research is the first attempt to apply mutation testing to mobile apps [66].

3.1.2 Mutation Testing for eXtensible Markup Language (XML)

Android applications use the eXtensible Markup Language (XML) intensively, from defining layouts, storing data, to configuring the system. The Android mutation technique in this research not only mutates an Android app’s Java source code, but also mutates its XML files. Several papers also studied mutation testing for XML.

To test messages transmitted between different web components, Lee and Offutt applied mutation analysis to XML data by defining web component Interaction Mutation Operators.
to mutate the interaction recorded in XML files [102]. Then, test cases are designed to detect the changes made to XML messages. The technique proposed by Lee and Offutt focuses on checking the semantic correctness of the interactions between web components. From the perspective of web applications, the web component interaction mutation operators do not modify original source code or structure of web applications, but create mutants of XML messages.

Similarly, Offutt and Xu approached the problem of input data validation for web services by designing mutation operators that modified XML schema [135]. 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. This research is slightly different in that, instead of defining input data, the mutated XML files are used to define GUI layouts and to configure the app. They are considered indispensable to the source code.

### 3.1.3 Reducing the High Cost of Mutation Testing

“Cost” in mutation analysis can be in different forms. According to Kurtz et al. [95], for software engineers and testers, they care more about how far away from obtaining a mutation adequate test set. When an engineer uses mutation testing to test his program, he usually first uses a mutation analysis tool to generate a set of mutants. Then, he chooses a mutant and designs a test to kill it. After that, he removes all mutants killed by this test. Alternatively, he may find that the mutant is not killable, i.e., an equivalent mutant, then removes it. When the engineer designs a test set that kills all the non-equivalent mutants, he obtains a mutation adequate test set. Therefore, from software engineers and testers’ perspective, the cost consists of the time and efforts used in designing tests and identifying equivalent mutants. Also, the paper of Kurtz et al. [95] defines *test completeness* as the ratio of tests designed to the total number of tests required to kill all non-equivalent mutants. Moreover, Kurtz et al. [94] identified that some mutants subsume others. This redundancy among mutants made test design and execution very expensive,
because engineers and testers must consider excessively more test requirements than actually necessary. The experiment in Section 5.4 also investigates the redundancy in Android mutation testing, but considers the redundancy among mutation operators instead of the redundancy among mutants.

From researchers’ perspective, computational time and efforts are the major cost, such as how much time is required to generate and kill all mutants. This study considers the computational time and effort as the major cost of Android mutation testing for executing tests against mutants. In addition, conducting mutation testing for Android apps has some extra cost that traditional mutation testing does not have. For example, because most Android emulators and mobile devices work much slower than personal computers, the computation time of killing mutants has to be inevitably prolonged. It is imperative to address the cost issue of mutation testing. Several papers have investigated saving the expensive cost of mutation testing.

Generally, traditional mutation testing uses three types of approaches to reduce the cost: do-fewer, do-smarter, and do-faster [134, 150]. As a do-fewer approach, selective mutation was introduced by Wong and Mathur to choose only a subset of mutation operators [157, 159]. The muJava tool selects 15 operators to preserve almost the same test coverage as non-selective mutation [114]. Additionally, empirical studies in both Java and C show that the Statement DeLetion (SDL) is able to result in very effective tests with much cheaper cost [63, 67]. Deletion operators are also included in the empirical study of this research.

3.1.4 Minimal mutation analysis and dominator mutation score

Ammann et al. [42] started by exploring the subsumption relationship among mutants. Subsumption is used to theoretically compare test criteria: a criterion C1 subsumes another criterion C2, if every test that satisfies C1 is guaranteed to satisfy C2 [43]. Kurtz et al. [96] defined subsumption relationship in mutation testing as: a mutant $m_1$ subsumes another mutant $m_2$ if a test that kills $m_1$ is guaranteed to kill $m_2$. Apparently, when designing and executing tests, $m_2$ becomes redundant since it does not contribute to the quality of the
tests. Therefore, Ammann et al. [42] defined a minimal set of mutants that does not have any redundant mutants. Minimal mutants are further renamed as dominator mutants in the paper of Kurtz et al. [94].

Since the presence of redundant mutants, Kurtz et al. [95] found a critical problem regarding mutation score, which is the ratio of killed mutants to the total killable mutants and the most widely used measurement in assessing the effectiveness of tests in mutation testing. A majority of researchers and engineers use mutation score as the indicator of how far away from obtaining a mutation adequate test set. Kurtz et al. [95] found that during the process of mutation analysis, the first few tests may kill a large portion of non-equivalent mutants, which makes the mutation score grow rapidly at the beginning. For example, a mutation score of 70% after first seven tests certainly does not indicate the engineer needs only three tests to achieve mutation adequacy. In contrary, killing these 30% unkilled non-equivalent mutants usually requires a significant amount of work. In other words, mutation score is inflated and is not linear. Thus, mutation score is not able to indicate test completeness.

After that, Kurtz et al. [95] defined the dominator mutation score as the ratio of the number of killed dominator mutants to the total number of dominator mutants. Their evaluation results showed that dominator mutation score is a better indicator in measuring test completeness than traditional mutation score. Furthermore, using dominator mutation score, Kurtz et al. [97] found that traditional selective mutation approaches were not suitable for all kinds of programs. Different sets of mutation operators should be selected for different programs. They recommended that a more specialized mutation operator selection strategy towards different programs should be designed. Along with this idea, recent research of René et al. [88] designed an approach of using a program’s abstract syntax tree to model program context information, and then predicting the usefulness of mutants towards the program.

However, a significant number of robust tests are necessary to conduct minimal mutation analysis and calculate dominator mutation score. Consequently, in this research, despite the flaw of mutation score, we have to continue using mutation score as the measurement
in the evaluations, due to the expensive cost of Android mutation testing, in terms of computational time and effort. A major future work of this research is to apply minimal mutation analysis and dominator mutation score in Android mutation testing.

3.2 Testing Android Applications

Android’s development environment includes its own test framework [10], which extends the ubiquitous JUnit. Additionally, several testing automation frameworks are available to testers. Many testers use Robotium [29] in unit testing, system testing, as well as user acceptance testing. It is also compatible with other code coverage measurement tools, such as Emma and Cobertura. Thanks to its APIs that directly interact with Android GUI components through run-time binding, people with little knowledge of implementation details can also write tests with Robotium. It is possible to test an app with Robotium even if only its Android Application Package (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. The preliminary work partly employed evoDroid [117], a Robotium test generation tool using evolutionary algorithms, to generate experimental test cases. However, the Android mutation technique in this research is available to accommodate all kinds of Android tests. Another framework for Android apps is Robolectric [28], which runs on the Java VM, instead of Dalvik or ART. It splits tests from the emulator, making it possible to run tests by directly referencing the Android library files. In testing Android apps, one challenge is the variety of hardware specifications, e.g., different screen sizes and resolutions. To address this, Selendroid [30] enables testers to distribute their tests across multiple emulators with different configurations. All these frameworks automate execution, but none supports test value generation, test criteria, or any other aspect of test design.

Several research papers focus on random test value creation. Considering an Android app as an event-driven system, Amalfitano et al. [39, 40] presented an approach that constructs a GUI model of the app under test by a code-crawling algorithm, and then randomly
fires events to generate test cases. As this technique is designed for crash testing and regression testing, it may be appropriate for detecting run-time crashes or exceptions. However, many faults do not propagate to crashes or exceptions. Li and Offutt [105] found that, in Java programs, only about 30% of software failures are runtime crashes. Even though there are no similar research findings for Android apps, only detecting crashes inevitably misses many faults. Moreover, some Android apps include Services or only execute at the background without a GUI. Constructing test cases based on GUI models may potentially overlook certain faults residing in Service or other components.

Hu and Neamtiu [83] conducted a fault study based on ten open source Android apps, and categorized Android faults into eight types: activity error, event error, dynamic type error, unhandled exceptions, API error, I/O error, concurrency error, and others. Assuming each type of fault should share a particular pattern in its log entries, they generate random GUI test inputs with the help of Android Monkey, and then collect execution logs from an instrumented Dalvik VM, which are further used to identify faults through different patterns. However, one premise of this approach is that the fault patterns in terms of log traces must be well-defined prior to the process. Without the patterns, people cannot locate potential faults in a huge pile of log traces. Moreover, elaborating all the possible patterns requires extensive analysis on a large number of apps, and is very less likely to be exhausted.

Also viewing Android apps as event-driven programs, Dynodroid [115] 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. Unlike other techniques, Dynodroid considers system events in addition to UI events to generate both human and machine inputs. To compute relevant events for test inputs, the Dynodroid paper also describes three event selection strategies: Frequency, UniformRandom, and BiasedRandom. An empirical study is conducted in this research to evaluate the fault detection effectiveness of Dynodroid.

Some researchers use model-based approaches to generate tests for Android apps. By
employing Android Monkey, TEMA [147] uses state machines (labeled state transition systems) to generate test sequences. However, two levels of state machines (action machine and refinement machine) need to be created by hand. MobiGUITAR [41] automates GUI-driven testing of Android apps by extracting run-time states of GUI widgets, and generates tests with the abstraction of models. Compared with Android Monkey and Dynodroid, MobiGUITAR was reported to detect more faults. ORBIT [160] creates a GUI model of the app and then generates tests. A3E [46] uses static taint analysis algorithms to build a model of the app, which is then used to explore the Activities in the app automatically. These papers focus on constructing models from which tests can be designed, as opposed to applying a test criterion such as mutation.

Some papers extend symbolic execution to test Android apps. Mirzaei et al. [124] created stubs and mock classes to make Android apps run on Java PathFinder (JPF) [20]. Merwe et al. [151, 152] developed JPF-Android by extending JPF to verify Android apps. However, the state explosion problem limited its ability to generate complex test inputs. Jensen et al. [85] 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. The Android mutation testing in this research tries to maximize test case effectiveness through mutation testing, an exceptionally strong coverage criterion. Anand et al. [44] used dynamic symbolic execution [93,132] in the form of concolic testing [74] to test an Android library. Their testing used pixel coordinates to identify valid GUI events.

Several papers applied evolutionary algorithms [116, 117], or multi-objective search-based software testing techniques [119] to test Android apps. These techniques focused on generating effective test inputs for GUI testing of Android apps, instead of using test criteria.

Choudhary et al. [55] evaluated seven automated Android test input generation tools with 68 open source Android apps. The evaluation assessed the tools from four perspectives: code coverage, fault detection effectiveness, compatibility, and usability. All 69 subjects were obtained from the evaluation of at least one tool, with the same version number as
in their original evaluation. This research uses a different approach to evaluating the fault detection effectiveness of different Android testing techniques. Naturally occurring faults and crowdsourced faults are used as the benchmarks. Since the benchmarks used in this research are significantly larger than Choudhary et al. used, with different natures of faults, our results are also different.

3.3 Android Permissions and Security Issues

Security of Android apps has been drawing much attention, one possible reason of which is the unique mechanism of Android permissions. The following papers investigated the usage of Android permissions.

Davi et al. [61] illustrated the privilege escalation attacks in Android apps with real examples, indicating that the existing model used by Android to enforce security policies is ineffective, and cannot withstand malicious attacks.

Felt et al. [73] introduced Stowaway to detect overprivileged Android apps via testing. By applying Stowaway to 940 Android apps, they found that one-third of Android apps requested extra unnecessary privileges. Apparently, many developers do not follow the principle of least privilege. When users casually accept dangerous permissions during installation, they drastically increase the risk of privacy leakage and security exploitation.

Pandita et al. developed Whyper [137], which uses Natural Language Processing (NLP) techniques to identify permissions required by Android apps according to their description. It not only helps users understand why an Android app requires a specific permission, but also provides developers feedback for modifying their programs.

Vidas et al. [154] presented a Permission Check Tool that can scan the source code of a given Android app, and provide the least permissions required by the app. Their results also show that, in addition to requesting extraneous permissions, some Android apps also specify fictitious permissions that have been obsolete for a very long time or do not exist at all.

All the techniques above are specifically designed to address the security of Android
However, a security vulnerability is also one kind of software fault, and should not be separated from other faults when testing. To enforce the principle of least privilege and provide comprehensive testing, this research designs an explicit, novel, mutation operator that is able to guide testers to explore whether the app under test has requested unnecessary privileges.

3.4 GUI Testing and Graphical Test Oracles

Many researchers consider Android apps to be event-driven systems, and test them in similar ways as traditional GUI applications. This is not enough because Android apps contain four types of components: Activity, Service, Content Provider, and Broadcast Receiver. Only Activity presents GUI on screen. Testing Android apps as GUI applications leaves the other three components insufficiently tested. Understanding commonly used technologies in testing GUI applications will help the tester design more effective approaches in testing Android apps.

Capture-replay tools are widely used to test GUIs. Testers record and capture event sequences in scripts that can be used to reproduce the same events to mimic human actions in subsequent testing activities, e.g., after adding new features or updating source code. Many research papers [52, 69, 146] proposed approaches like this. Specifically for testing Android apps, Android Capture and Replay Testing Tool (ACRT) [109] and REcord and Replay for ANdroid (RERAN) [75] are based on the capture-replay process. Different from other GUI capture-replay testing tools, which cannot solve the problem of sophisticated hand input events such as tapping, zooming, and swiping, RERAN addresses the challenge by directly capturing and replaying low-level events on an Android device. Moreover, it is able to record inputs at the system level from multiple sensors in the device, such as accelerometers and compasses.

Another type of GUI testing approach is model-based test generation. This approach first constructs models of GUI components and interactions. Then, tests are generated based
on models by applying different algorithms. GUITAR [127] is a typical automated model-based framework that provides comprehensive GUI test generation. To expand GUITAR to the domain of mobile applications and address the challenges in testing Android apps, Amalfitano et al. proposed MobiGUITAR [41]. It models Android apps’ GUI widgets into state machines, and generates tests by applying graph testing coverage criteria. Results showed that MobiGUITAR detected more faults than Android Monkey and Dynodroid can. However, when the model of the apps under test grows rapidly, there might be difficulties in handling a large number of tests. Also, GUI widgets are not the only types of inputs in Android apps. An app may also accept user or system events, such as GPS locations, environment temperature, and screen orientation, etc. Test cases derived from GUI models might not provide these inputs.

Most techniques generate test inputs, but not test oracles. This means that many approaches can only detect crashes or exceptions, which has been called the Null Test Oracle Strategy (NOS). Li and O’Fallt [104] analyzed different test oracle strategies and found that NOS can only detect around 30% of faults, indicating that more than two-thirds of the faults did not lead to crashes thus remained undetected. Instead of focusing on generating test inputs, some researchers propose to address the issue of test oracles. Employing computer vision techniques, Lin et al. [106] introduced the SmartPhone Automated GUI Testing tool with Camera (SPAG-C) to improve the efficiency and reusability of test oracles for Android apps. Generally, it is still a capture-replay process. However, in the capture phase, SPAG-C records screenshots from an external camera; while in the replay phase, SPAG-C conducts image comparison on the captured screenshots with computer vision algorithms including Speeded Up Robust Features (SURF) [50], template matching [32], and histogram [138]. Experimental results show that it provides efficiency and reusability by significantly shortening the time to re-design tests. However, using external cameras to capture screenshots inevitably downgrades the accuracy of data.
3.5 Mining Source Code Repositories and Bug Reports

Mutation testing is called a fault-based technique because most mutants mimic faults that occur during software development, to challenge testers to design effective test cases. Thus, to design effective mutation operators, this research analyzed change history logs of Android source code in open-source project repositories to discover fault fixing activities and to categorize common faults in Android projects. In addition, to evaluate the fault detection effectiveness, this research collects naturally occurring faults by mining open source Android apps’ repositories. Several papers also investigated source code repositories, as well as their corresponding bug reports.

Instead of finding bug patterns, Coelho et al. [57] mined repositories of Android open source projects to identify bug hazards, which were defined as circumstances that may lead to faults. They analyzed more than 6,000 exception stack traces from 639 Android projects hosted on Github and Google Code. Their results showed that Null Pointer Exception (NPE) is the most reported exception thrown by Android apps, and around 52% of mined projects have at least one failure with NPE thrown. Their results conform to the findings of this research.

Bug reports often describe what failure has happened without giving specific details on where and what the fault is. Developers have to spend a lot of their time to reproduce the scenario and locate the fault. Zhou et al. [161] introduced BugLocator, a tool that can retrieve relevant files from bug reports. It searches the source code repository, finds past similar bug fixes, and ranks source code files based on their relevance. For each fault, BugLocator predicts several possible faulty files. An experiment showed that more than 60% of faulty files were successfully identified in the top ten possible faulty files predicted by BugLocator.

Gyimesi et al. [80] proposed to generalize characteristics from old faults to predict possible future faults. To look for faults, the authors designed a tool to analyze thirteen Java projects on GitHub and collect code change history associated with their bug reports. However, the paper does not describe faults located with the method.
Dallmeier and Zimmermann [58] created a repository called \textit{iBUGS} that extracts bug reports, faulty versions and their fixes, and associated tests from software repositories. With their experiment on project ASPECTJ, one critical finding was that more than 54\% of all faults were fixed within a single method, and there were a large number of one-line faults. In mutation testing, a fault-based testing technique, mutation operators apply a single syntactical change to the program under test. The results of \textit{iBUGS} provide experimental evidence to support the methodology of mutation testing.

3.6 Crowdsourcing in Software Engineering

Crowdsourcing is the act of using an open call to recruit an undefined group of professionals and assign them tasks [118]. This research introduced a crowdsourcing approach to collect software faults in Android apps to evaluate the fault detection effectiveness of Android testing techniques.

LaToza and van der Hoek[99] categorized three crowdsourcing models in software engineering:

1. \textit{Peer production}, where contributors perform tasks that are controlled in a decentralized manner

2. \textit{Competitions}, where contestants perform tasks but are only rewarded if they end up within a success threshold

3. \textit{Microtasking}, where participants perform tasks that are intended to be quick, small and self-contained to achieve scalability

Because creating a fault in software programs is a microtask for software developers, this research employs the \textit{Microtasking} model to collect crowdsourced faults. Crowdsourcing in software engineering is becoming more commonly used in different areas, such as testing programs and fixing faults. However, as far as we know, this research is the first attempt to use crowdsourcing to create software faults in software testing empirical studies.
LaToza et al. [98] introduced a new software development process, called crowd development, which organizes software development tasks based on microtasks. It has the potential to reduce the costs of software development, as well as increase the productivity of programmers. In another paper, LaToza et al. [100] developed a novel approach that applied microtask crowdsourcing in software development by decomposing programming work into microtasks. Moreover, the empirical study showed that it is very feasible to apply crowdsourcing in developing software.

Liu et al. [110] applied crowdsourcing to usability testing, and compared their crowdsourcing approach with a lab usability test. They found that crowdsourcing was faster, cheaper, and easier to conduct than standard lab-based usability testing. However, due to the diverse background of participants, careful task design and background review are necessary to the success of crowdsourcing approaches.

Dolstra et al. [68] introduced a crowdsourcing approach to performing GUI tests through the Internet. They created a website that enabled testers to access the virtual machines that contain the GUI software under test, so that these testers can test the software using their own web browsers. Their experiment showed that it was very feasible to apply crowdsourcing approach in GUI testing over the Internet, even though some participants suffered from network connection issues.
Chapter 4: Mutation Testing for Android Applications

This chapter introduces and describes Android mutation testing, including the novel Android mutation operators designed in this research.

4.1 Mutating Android Applications

Mutation analysis cannot be directly applied to Android apps the same way it is on traditional Java programs, because Android apps have different programming structure and are developed, installed, and tested in different ways. Traditionally, Java mutation analysis tools either mutate the Java source code and compile to bytecode class files, or first compile to bytecode, then change the bytecode. After that, the Java bytecode files are dynamically linked by the language system during execution. However, Android apps have an 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. Moreover, since Android apps intensively employ XML files for program configuration, layout design and specification, this research designs Android mutation operators to mutate XML files as well. These factors significantly change the process, design, and implementation of Android mutation testing tools.

Figure 4.1 illustrates how the Android mutation analysis engine works. Below are the steps for conducting mutation analysis on Android apps. Note that steps 3, 4, 6, and 7 are different from traditional Java mutation testing processes.

1. The user first selects which mutation operators to use. This research designed and built an Android mutation analysis tool, called muDroid, which includes seventeen new Android mutation operators, fifteen traditional Java mutation operators from muJava [114], and four deletion operators [63, 67]. The Android mutation analysis
Figure 4.1: Performing Mutation Analysis on Android Apps

tool extends part of the muJava [114] mutant generation engine to implement these mutation operators.

2. For the traditional Java mutation operators, the Android mutation analysis tool changes the original Java files according to defined mutation rules, and compiles them to bytecode.
3. (New to Android mutation.) XML mutation operators are applied directly to XML files, creating new copies of each file as XML mutants. Then, the Android mutation analysis tool swaps the mutated file into place to prepare it for dynamic binding when building APK files.

4. (New to Android mutation.) The mutation system selects a mutated Java bytecode class file or XML file, incorporates other project files, and generates a mutated APK file as a mutant of the Android app under test. Some mutants might cause compilation errors, i.e., stillborn mutants, while generating APK files. These stillborn mutants are discarded immediately and not used in the final results.

5. The Android testing framework extends JUnit [21] to support the testing of Android apps [10]. In addition, several external Android testing automation frameworks, such as Robotium [29], Espresso [15], and Selendroid [30], are frequently used by testers to provide automated testing for Android apps. The Android mutation analysis tool accommodates all these Android testing automation frameworks. Test cases can be designed by testers with Android testing automation frameworks to kill mutants, or a set of externally created test cases, such as tests from other automated test generation tools, can be used.

6. (New to Android mutation.) 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. (New to Android mutation.) Then, each APK mutant file 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. To save the expensive execution cost, the Android mutation analysis tool can connect to an unlimited number of
Android emulators and mobile devices, and perform execution in parallel.

8. After collecting all the results, the Android mutation analysis tool 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 Android mutation analysis tool computes the mutation adequate score, i.e., the ratio of the number of the mutants killed by the tests to the total number of non-equivalent mutants. The tool does not implement any heuristics to help identify equivalent mutants, so the tester needs to eliminate equivalent mutants manually.

4.2 Android Mutation Operators

After describing the process of performing mutation analysis on Android apps, this section introduces the Android mutation operators designed in this research. Mutation operators are rules that specify how to make changes to program source code or software artifact. Well-designed mutation operators lead to very powerful tests, but poorly-designed operators sometimes may result in ineffective tests. Mutation operators are designed in one of the two ways:

1. If pre-defined fault models are available, each type of the fault in the fault models can be used to define a mutation operator that can create instances of the fault. For example, the class-level mutation operators in muJava [114,133] were designed according to the fault model created by Offutt et al. [130].

2. An alternative 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 a pre-defined fault model does not exist, this research first conducted a fault study to identify common faults in Android app development. Some Android mutation operators
Table 4.1: Android Mutation Operators

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

were designed according to these common faults. This research then analyzed unique characteristics and features of Android apps, and identified challenges in testing Android apps. Additional Android mutation operators were designed to address these characteristics and challenges. Consequently, this research used both ways to design Android mutation operators.

Table 4.1 lists the seventeen Android mutation operators designed in this research. The goal of these Android mutation operators is to provide a sophisticated testing technique that can provide comprehensive testing for Android apps and address the unique testing challenges identified in Section 1.2. These Android mutation operators fall into seven categories: event-based, component lifecycle, XML-related, common faults, context-aware, energy-related, and network-related.
4.2.1 Event-based Mutation Operators

Android apps are event-based programs that implement event handlers to recognize and respond to various events that are initiated by different user actions. Usually, clicking and touching are the most common actions when people use mobile devices. Intent objects are used to facilitate the communication between Android components. They are abstractions of different operations to be performed by Android components. Usually, Intent objects carry data and other information that are required by the target component, and considered as the key to event-driven programs, i.e., Android apps.

Thus, this section introduces two Android mutation operators for event handlers, OnClick Event Replacement (ECR) operator and the OnTouch Event Replacement (ETR) operator, and two for Intent objects, Intent Payload Replacement (IPR) and Intent Target Replacement (ITR).

1) Intent Payload Replacement (IPR)

Definition:

Each payload of an Intent object is replaced by a pre-defined default value (Table 4.2) according to the data type of the payload. Each primitive type payload is replaced by the value zero. Each boolean payload is replaced by both true and false. Each String payload is replaced by empty strings and null values. Each Array payload or other type of payload is replaced by null values cast into the appropriate type.

Restriction:

1. IPR is not applied to implicit Intent objects.

An Intent object can carry different types of data (called payload) in the form of key-value pairs. For example, in Figure 4.2 the original method has an Intent object that is initialized to send to DisplayMessageActivity.class. It also carries the String value of message with a key as EXTRA_MESSAGE. The putExtra() method takes the key name as the first parameter, and the value as the second parameter. When the target Activity DisplayMessageActivity.class receives the Intent, it is able to load the data with the same
Figure 4.2: Intent Payload Replacement Mutation Operator

The IPR operator mutates the second parameter to the default value of the underlying data type, as listed in Table 4.2. Objects with primitive numeric types, including int, short, long, float, double, and char, are replaced by the value zero, and boolean variables are replaced by both true and false. String objects are replaced by empty strings and null values. Arrays and other types of objects are replaced by null values cast into the appropriate types.

Figure 4.2 shows how IPR works. The payload of the Intent object, the String object message, is replaced with an empty String. An IPR mutant can only be killed by a test that verifies the value received through an Intent object is correct.

2) Intent Target Replacement (ITR)

Definition:

The target of an Intent object is replaced by each of the other compatible classes in the
Table 4.2: IPR Default Values

<table>
<thead>
<tr>
<th>Original Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, short, long, float, double, char</td>
<td>0</td>
</tr>
<tr>
<td>boolean</td>
<td>true / false</td>
</tr>
<tr>
<td>String</td>
<td>“” / (String) null</td>
</tr>
<tr>
<td>Array</td>
<td>(Array) null</td>
</tr>
<tr>
<td>Others</td>
<td>(Others) null</td>
</tr>
</tbody>
</table>

same package.

Restrictions:

1. ITR is not applied to implicit Intent objects.

2. An Intent target is not replaced by itself.

There are two different ways of using Intent objects in Android apps: implicit Intent and explicit Intent [7]. An implicit Intent declares a general event to perform without indicating a specific target, and allows other apps to handle the event. Because this research does not consider inter-app communication, implicit Intents are out of the scope of this research. Conversely, an explicit Intent needs to specify which component should be started by declaring the Intent with the target component’s name within an app.

Figure 4.3 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 compatible classes. ITR forces the tester to design test cases that check that the target activity or service is launched successfully after the Intent is executed.

3) OnClick Event Replacement (ECR)

Definition:

Each OnClick event handler is replaced by each of the other compatible OnClick event handlers in the same class.

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

A. Original

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

B. Mutant

Figure 4.3: Intent Target Replacement Mutation Operator

1. ECR does not replace anything if there is only one OnClick event handler in the class, i.e., no other compatible OnClick event handler exists.

2. A OnClick event handler is not replaced by itself.

Events in Android apps are usually used to respond external user actions, such as interactions (e.g., clicking, touching, and dragging) with touch screens, particularly for smartphones and tablets. The Android operating system places every event into a first-in-first-out queue. Android apps must implement event handlers to respond to particular events and fulfill the required functionality.

ECR first searches and stores all the event handlers that respond to OnClick events in the current class. Then, it replaces each handler with every other compatible handler collected previously. Figure 4.4 shows an example ECR mutant. The original program has two event handlers: \texttt{incrementPrepTime()} for the button \texttt{mPrepUp}, and \texttt{decrementPrepTime()} for the button \texttt{mPrepDown}. In the mutant, ECR replaces the event handler \texttt{incrementPrepTime()} with \texttt{decrementPrepTime()}. To kill ECR mutants, each widget’s OnClick event has to be executed by at least one test. Also, the expected action of the event needs to be verified by the test oracle.

4) \textbf{OnTouch Event Replacement (ETR)}
mPrepUp.setOnClickListener (new OnClickListener() {
    public void onClick (View v) {
        incrementPrepTime();
    }
});
mPrepDown.setOnClickListener (new OnClickListener() {
    public void onClick (View v) {
        decrementPrepTime();
    }
});

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

A. Original

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

B. Mutant

Figure 4.4: OnClick Event Replacement Mutation Operator

Definition:

Each OnTouch event handler is replaced by each of the other compatible OnTouch event
handlers in the same class.

Restrictions:

1. ETR does not replace anything if there is only one OnTouch event handler in the
class, i.e., no other compatible OnTouch event handler exists.

2. A OnTouch event handler is not replaced by itself.

Touch actions happen when the user places one or more fingers on the screen of mobile
devices. Once the last finger leaves the screen, the Android app interprets the entire finger
movement pattern into a particular gesture, such as tapping, swiping, and zooming in or out. Figure 4.5 shows an example ETR mutant. The original program responds to touch actions on *myLayout* by capturing the vertical and horizontal coordinates on the screen and processes based on the coordinates. Similarly, ETR replaces the event handlers for each OnTouch event with other collected available event handlers. In Figure 4.5, the event handler in *myLayout* is replaced by another event handler in *yourLayout*.

```java
myLayout.setOnTouchListener (new OnTouchListener()
{
    public void onTouch (View v, MotionEvent event) {
        int x = (int)event.getX();
        int y = (int)event.getY();
        handleTouch(x, y);
    }
});
yourLayout.setOnTouchListener (new OnTouchListener()
{
    public void onTouch (View v, MotionEvent event) {
        handleTouch(event);
    }
});
```

**A. Original**

```java
myLayout.setOnTouchListener (new OnTouchListener()
{
    public void onTouch (View v, MotionEvent event) {
        handleTouch(event);
    }
});
yourLayout.setOnTouchListener (new OnTouchListener()
{
    public void onTouch (View v, MotionEvent event) {
        handleTouch(event);
    }
});
```

**B. Mutant**

Figure 4.5: OnTouch Event Replacement Mutation Operator
4.2.2 Component Lifecycle Mutation Operators

Section 1.2 describes the unique testing challenge brought by the pre-specified lifecycle of Android components. Inappropriately handling the unique lifecycles of Android components is a common mistake in Android app development. For example, Figure 1.5 showed the lifecycle of Activity components. Seven methods are used to fulfill transitions among different states in the Activity lifecycle. Figure 1.6 illustrated the lifecycles of Service components. Fewer methods are used to facilitate the transitions in the Service lifecycles than the Activity lifecycle, but Service components have two types: unbounded and bounded. Additionally, these two types are not mutually exclusive. For example, a bounded Service can also be started with the \texttt{onStartCommand()} method. Thus, this research designs two Android mutation operators to modify the methods in the lifecycles of different Android components.

5) Lifecycle Method Deletion (MDL)

\textbf{Definition:}

Each Activity lifecycle method is deleted.

\textbf{Restriction:}

1. MDL ignores the method that only has one statement that calls the overridden method in its super class.

When implementing an Activity, developers need to override the methods in the Activity lifecycle to define different states and transitive operations among the states. As the example in Section 1.2 illustrates, correctly implementing these methods is critical to the smooth execution flow of the Activity. 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 [114], but only considers the methods related to the Activity lifecycle. Figure 4.6 shows an example MDL mutant. MDL removes the implementation inside the original \texttt{onPause()} method. Then, the Android system needs to
call the `onPause()` method in the super class. To kill this MDL mutant, the tester needs to design tests to force the Activity switch among different states through the mutated transition methods.

```
@Override
public void onPause()
{
    super.onPause();
    SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
    SharedPreferences.Editor editor = settings.edit();
    editor.putLong(PREP_SECONDS, getPrepTime());
    editor.putLong(MEDITATION_MINUTES, getMeditateTime());
    editor.commit();
    ... ...
}
```

A. Original

```
@Override
public void onPause()
{
    super.onPause();
}
```

B. Mutant

Figure 4.6: An Example MDL Mutant

6) **Service Lifecycle Method Deletion (SMDL)**

**Definition:**

Each Service lifecycle method is deleted.

**Restrictions:**

1. If the method calls the overridden method in its super class, SMDL will keep the statement of this call.

2. If the lifecycle method returns boolean values, SMDL will force the method to return `true` and `false`.

Service components are invisible to the user and always stay in the background for a longer period of the time than Activity components, to perform long-term running tasks.
As illustrated in Figure 1.6, the lifecycle of Service components are also different from the lifecycle of Activity components. The implementation of Service components also requires accommodating the lifecycles appropriately. Otherwise, the Android app that uses the Service may behave incorrectly.

SMDL deletes each lifecycle method in Service components, including `onCreate()`, `onStartCommand()`, `onBind()`, `onRebind()`, `onUnbind()`, and `onDestroy()`, to force the Android operating system to call the version in the super class, or an empty class without implementation. Figure 4.7 shows an example SMDL, in which SMDL disables the implementation in the `onUnbind()` method. When a client requests to unbind the service, the expected behavior in the service will be skipped. To kill an SMDL mutant, the tester must design a test that ensures the Service works as expected.

```java
@Override
public boolean onUnbind(Intent intent) {
    if (pendingAlarms.size() == 0) {
        stopSelf();
        return false;
    }
    return true;
}
```

A. Original

```java
@Override
public boolean onUnbind(Intent intent) {
    return true;
}
```

B. Mutant

Figure 4.7: An Example SMDL Mutant
4.2.3 XML-related Mutation Operators

As mentioned in Section 1.2, Android apps use many XML files, not just the manifest file. XML files are used to define user interfaces, to store configuration data such as permissions, to set the default launch activity, and more. XML files form one of the key challenges in testing Android apps. Other Android testing techniques do not target XML files. However, visual appearance, usually rendered from XML files, is categorized as a common Android app fault. This research designs five novel XML-related Android mutation operators. These operators are unusual in that they do not modify executable code, but static XML.

7) Button Widget Deletion (BWD)

Definition:
Each Button widget is deleted from the UI.

Restriction:
1. None.

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

Figure 4.8 shows how the example BWD mutant in Figure 4.9 was implemented. Since removing a button from the XML layout specification file will result in compiling errors, in this research, BWD inserts an extra XML attribute, \texttt{android:visibility=“gone”}, into the button declaration, to delete the button without incurring any compiling errors.

8) EditText Widget Deletion (TWD)

Definition:
Each EditText widget is deleted from the UI.

Restriction:
Figure 4.8: An Example BWD Mutant
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 4.9 shows an example TWD mutant where the bill amount cannot be displayed. Similar to the BWD example shown in Figure 4.8, TWD inserts an extra XML attribute, `android:visibility="gone"`, into the EditText declaration, to delete the EditText widget. To kill this mutant, a test must use the bill amount.

9) **Activity Permission Deletion (APD):**

**Definition:**

Each permission of the Android app under test is deleted from its manifest file.

**Restriction:**

1. None.

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 (Android-Manifest.xml). Some apps aggressively request unnecessary, even irrelevant, permissions, and many users simply click “OK” without paying attention to the details of these requested permissions when installing an app. This may create security vulnerabilities to Android systems.

APD mutants delete an app’s permissions from its AndroidManifest.xml file, one at a time. If this mutant cannot be killed by any test, it means that the app asked for a permission it did not need. For example, in Figure 4.10, the original program requests four permissions: WRITE_SETTINGS, WAKE_LOCK, MODIFY_AUDIO_SETTINGS, and VIBRATE. APD deletes the VIBRATE permission in the example mutant. Then, the app is not allowed to use the device’s vibrator. A killing test for this mutant must lead the app to attempt to access the vibrator of the Android system.
10) **Button Widget Switch (BWS)**

**Definition:**

The locations of each pair of button widgets on the same screen are switched.

**Restriction:**

1. A button widget is not switched with itself.

It is common for testers to design test cases to ensure an app works as expected with respect to its functional requirements, and evaluate the GUI structure as a secondary issue. However, Android apps are event-based, which means it is essential to display the GUI structure appropriately, as well as handle user events. Unlike BWD, BWS does not remove a button widget, but switches the locations of two buttons on the same screen. In this way, the function of a button is unaffected, but the GUI layout looks different from the original version. BWS requires the tester to design tests that deliberately check the location (either relative or absolute) of a button widget. Figure 4.11 illustrates an example BWS mutant. The mutant on the right side switches the locations of button “7” and “OK.” Figure 4.12 shows how BWS switches the buttons in the XML layout specification file.

11) **TextView Deletion (TVD)**

**Definition:**

Each TextView widget is deleted from the UI.

**Restriction:**

1. None.

Android apps use TextView widgets to display text to users. Unlike EditText widgets, TextView widgets usually cannot be edited by users. The left screenshot in Figure 4.13 is from an Android app. “Subtotal,” “Tip (15.0%),” “Total,” and their numbers are all TextView widgets. As can be seen, unlike the Button and EditText widgets, TextView widgets usually do not associate with any user events, nor require any event handlers from the implementation of the app. However, TextView widgets are widely used by developers to present essential information.
TVD deletes TextView widgets from screens one at a time. Killing the TVD mutants needs tests to ensure that every TextView widget displays the correct information. The right side of Figure 4.13 shows an example TVD mutant in which the TextView widget of the total amount is deleted. Similar to BWD and TWD, TVD also inserts an extra XML attribute, \textit{android:visibility=“gone”}, to remove a TextView widget from the screen. To kill a TVD mutant, the tester needs to design a test to check whether the total amount is correctly displayed.

4.2.4 Common Faults Mutation Operators

Two mutation operators are designed based on common faults found in GitHub repositories.

12) \textbf{Fail on Null (FON)}
A. Original

B. Mutant

Figure 4.12: An Example BWS Mutant
Definition:
A special “fail on null” statement is inserted before each object is referenced.

Restriction:
1. None.

According to Arlt et al. [45], NullPointerException is one of the most common exceptions thrown in programs. A common cause is that developers sometimes forget to check if an object is null before accessing it. In the initial study on GitHub repositories, within 80 corrections to one app, 52 were patching null-checking statements. FON mutants add a “fail on null” statement before each object is referenced. For String objects, FON also adds a “fail on empty” statement before objects are accessed. Figure 5.5 shows an example FON mutant. The mutated statement is inserted before accessing members. FON mutants are
used to seek test cases that can make members null and trigger the “fail on null” statement.

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

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

Figure 4.14: Fail on Null Mutation Operator

13) **Orientation Lock (ORL)**

**Definition:**

A special screen-locking statement is inserted into every Activity to lock the screen orientation to be in portrait and landscape.

**Restriction:**

1. ORL is not applied to the Activity that has been implemented with a feature of freezing the screen orientation according to its design or requirements.

Mobile devices such as smartphones and tablets have the unique feature of being able to change the screen orientation. Thus, many apps change the layout of the GUI when the orientation changes. For example, YouTube automatically switches to play video in full screen when the orientation is changed from portrait to landscape. However, Android devices are manufactured by different factories with various hardware specifications, using different screen sizes and resolutions. This makes switching the orientation difficult for the developers, in turn leading to many faults in Android apps.
Figure 4.15 shows a correct and a faulty version of TippyTipper with different orientations. Even though both devices properly display the GUI in portrait orientation, when switching to landscape orientation, as shown in Figure 4.15, the user is not able to see or click the button at the bottom or scroll down the screen.

ORL mutants freeze the orientation of an activity to be in portrait or landscape, by inserting a locking statement into the source code. This locking statement calls the orientation API in the Android system. Figure 4.16 shows two examples of ORL mutants, in which the mutant_1 (Figure 4.16.B) fixes the screen orientation to portrait, and the mutant_2 (Figure 4.16.C) freezes the screen orientation to landscape. Only test cases that explicitly changes the orientation and checks whether the GUI structure is displayed as expected in both orientations can kill these mutants.

14) Fail on Back (FOB)

Definition:

A special “Fail on Back” event handler is inserted into every Activity to wait for being triggered after the user presses on the Back button.

Restriction:

1. FOB is not applied to the Activity that has been implemented with a feature of any event handlers for the Back button.

Section 1.2 introduced testing challenges caused by the Android system buttons. At the
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    View btn_one = findViewById(R.id.btn_one);
    ButtonOne buttonOne = new ButtonOne();
    btn_one.setOnClickListener(buttonOne);

    View btn_two = findViewById(R.id.btn_two);
    ButtonTwo buttonTwo = new ButtonTwo();
    btn_two.setOnClickListener(buttonTwo);
    ... ...
}

A. Original

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);

    View btn_one = findViewById(R.id.btn_one);
    ButtonOne buttonOne = new ButtonOne();
    btn_one.setOnClickListener(buttonOne);

    View btn_two = findViewById(R.id.btn_two);
    ButtonTwo buttonTwo = new ButtonTwo();
    btn_two.setOnClickListener(buttonTwo);
    ... ...
}

B. Mutant_1

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);

    View btn_one = findViewById(R.id.btn_one);
    ButtonOne buttonOne = new ButtonOne();
    btn_one.setOnClickListener(buttonOne);

    View btn_two = findViewById(R.id.btn_two);
    ButtonTwo buttonTwo = new ButtonTwo();
    btn_two.setOnClickListener(buttonTwo);
    ... ...
}

C. Mutant_2

Figure 4.16: Two Example ORL Mutants
application level, the Back button is the most impactful of the three system buttons. Unlike the Home and Recents buttons, which pause and terminate the app, the Back button lets users move back to previous screens, similar to the back button in web browsers. Many testers overlook its impact. A common Android failure is an app crashing when the Back button is clicked. Figure 4.17 shows an example FOB mutant. FOB injects a “Fail on Back” event handler into every Activity class. To kill FOB mutants, testers need to design tests that press the Back button at least once at every Activity.

4.2.5 Context-Aware Mutation Operator

15) Location Modification (LCM)

Definition:

The values of latitude and longitude attributes of each location variable are incremented and decremented by one degree. The value of altitude attribute of each location variable is elevated and lowered by one meter. The value of speed attribute of each location variable is accelerated and decelerated by one meter per second.

Restriction:

1. None.

As stated in Section 1.2, 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 rarely used by Android apps (such as gravity and pressure). Consequently, we only considered 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. Figure 4.18 shows four example LCM
public class Total extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.total);
        ... ...
    }

    @Override
    public void onStart()
    {
        super.onStart();
        RefreshBillAmount();
        ... ...
    }
    ... ...
}

public class Total extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.total);
        ... ...
    }

    @Override
    public void onStart()
    {
        super.onStart();
        RefreshBillAmount();
        ... ...
    }
    ... ...
    public void onBackPressed()
    {
        fail();
    }
}

Figure 4.17: An Example FOB Mutant
mutants. To kill an LCM mutant, the tester needs to design tests to ensure the app behaves as expected at different locations.

### 4.2.6 Energy-Related Mutation Operator

Section 1.2 introduced the testing challenge regarding limited battery power of Android devices, and *energy bugs*, which are poor programming practices that consume extra energy. Guo et al. found that several of these bugs were caused by resource leaks, that is, resources acquired too early or released too late that deplete a device’s battery. One common resource leak is inappropriately using wake locks [79]. A *wake lock* is a mechanism used by the Android system to keep Android devices from going into sleep mode. An app needs to request appropriate wake locks if it requires certain system resources.

Figure 4.19 compares two energy consumption diagrams from the same app. Initially, the app has an *energy bug* (top), which is causing a performance failure. Hence, it is consuming an unexpectedly large amount of energy after its state transition from active to idle. An app is expected to be using 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 also in the background (e.g., any media player or service). After fixing the bug, the app’s energy consumption drops when entering the Idle state.

16) **WakeLock Release Deletion (WRD)**

**Definition:**

Each call to the `release()` method to release a wake lock is deleted.

**Restriction:**

1. None.

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 [145] for detecting and correcting *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.
private String currentLocation(int formatting)
{
    Location location = locationManager.getLastKnownLocation( best );

    if (location != null) {
        double latVal = location.getLatitude();
        double longVal = location.getLongitude();
    }
    ... ...
}

private String currentLocation(int formatting)
{
    Location location = locationManager.getLastKnownLocation( best );
    \texttt{location.setLatitude( location.getLatitude() + 1 );}
    if (location != null) {
        double latVal = location.getLatitude();
        double longVal = location.getLongitude();
    }
    ... ...
}

private String currentLocation(int formatting)
{
    Location location = locationManager.getLastKnownLocation( best );
    \texttt{location.setLongitude( location.getLongitude() + 1 );}
    if (location != null) {
        double latVal = location.getLatitude();
        double longVal = location.getLongitude();
    }
    ... ...
}

private String currentLocation(int formatting)
{
    Location location = locationManager.getLastKnownLocation( best );
    \texttt{location.setAltitude( location.getAltitude() + 1 );}
    if (location != null) {
        double latVal = location.getLatitude();
        double longVal = location.getLongitude();
    }
    ... ...
}

private String currentLocation(int formatting)
{
    Location location = locationManager.getLastKnownLocation( best );
    \texttt{location.setSpeed( location.getSpeed() + 1 );}
    if (location != null) {
        double latVal = location.getLatitude();
        double longVal = location.getLongitude();
    }
    ... ...
}

Figure 4.18: Four Example LCM Mutants
consumption, both statically and dynamically. We used this idea to design the WRD mutation operator, which deletes each call to the `release()` method 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. Figure 4.20 shows an example WRD mutant, in which WRD deletes the call to the `release()` method in the `onPause()` method. This example mutant mimics the common fault that developers forget to release the wake lock after putting the app to the background, resulting the app keeps draining the battery even at the idle state.

Testers can kill WRD mutants in one of two ways. First, Android SDK has a tool called
@Override
protected void onPause()
{
    super.onPause();
    if(sleepPref == true)
    {
        wakelock.release();
    }
}

A. Original

@Override
protected void onPause()
{
    super.onPause();
    if(sleepPref == true)
    {
        // wakelock.release();
    }
}

B. Mutant

Figure 4.20: An Example WRD Mutants
dumpsys that can capture system information from Android devices. Using this tool, testers can identify active wake locks after the app under test has been closed. Figure 4.21 shows part of the output from the dumpsys tool after launching the experiment subject JustSit, that is, when the app is in its active state. According to the output, JustSit has an active wake lock in the system. If the wake lock is handled properly, closing the app should release it successfully, that is, after it transitions to its idle state. If the wake lock is still active after the app is terminated, the WRD mutant is killed.

<table>
<thead>
<tr>
<th>Wake Locks: size = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
</tr>
<tr>
<td>SCREEN_DIM_WAKE_LOCK</td>
</tr>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
</tr>
</tbody>
</table>

Figure 4.21: Identifying Wake Locks in the Android System

Alternatively, testers can use external tools to profile and compare the energy consumption of the original app and the mutant apps. If testers can identify irregular battery drainage, the mutants are killed. For simplicity, we presented coarse active and idle states. More realistic scenarios would have several transitions between these states. To kill a WRD mutant, the tester must design tests that evaluate the energy consumption of the app.

4.2.7 Network-related Mutation Operator

The challenge discussed in Section 1.2 urges testers to consider different network connections when designing tests for their apps. This section introduces a network-related mutation operator that specifically helps address this challenge.

17) Wi-Fi Connection Disabling (WCD)

Definition:

Each Activity is inserted a special statement to disable the Wi-Fi connection.

Restriction:

1. None.
WCD inserts a special piece of source code into every Activity to disable the Wi-Fi connection when the Activity is launched. Figure 4.22 shows an example WCD mutant, in which the Wi-Fi connection is disabled by the special piece of source code inserted. WCD mutants mimic the scenario that the Android 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. Note that many Android apps do not have features that require network connections, which makes WCD mutants equivalent. However, identifying this type of equivalent WCD mutants is very straightforward and not time-consuming.

4.2.8 Summary

This chapter introduced 17 Android mutation operators. They are designed to test all identified unique features and novel characteristics of Android apps in this research, and to mimic common software programming faults discovered during the fault study. Based on the results in Chapter 5, these 17 Android mutation operators provide comprehensive testing for Android apps. Some of these operators are further refined in Chapter 5, to improve the effectiveness and efficiency of the initial set of mutation operators.
@Override
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  View btn_one = findViewById(R.id.btn_one);
  ButtonOne buttonOne = new ButtonOne();
  btn_one.setOnClickListener(buttonOne);

  View btn_two = findViewById(R.id.btn_two);
  ButtonTwo buttonTwo = new ButtonTwo();
  btn_two.setOnClickListener(buttonTwo);
  ...
 ...
}

A. Original

@Override
public void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  WifiManager wifiManager = (WifiManager) this.getSystemService(
      android.content.Context.WIFI_SERVICE);
  wifiManager.setWifiEnabled(false);

  View btn_one = findViewById(R.id.btn_one);
  ButtonOne buttonOne = new ButtonOne();
  btn_one.setOnClickListener(buttonOne);

  View btn_two = findViewById(R.id.btn_two);
  ButtonTwo buttonTwo = new ButtonTwo();
  btn_two.setOnClickListener(buttonTwo);
  ...
 ...
}

B. Mutant

Figure 4.22: An Example WCD Mutant
Chapter 5: Experiments

This chapter first introduces the functionality and design of the Android mutation analysis tool implemented in this research, and then describes three experiments. First is an evaluation of the feasibility of using mutation testing on Android apps, second is a study to evaluate the effectiveness of Android mutation testing, and third is a study to measure redundancy among Java and Android mutation operators when testing Android apps.

5.1 Android Mutation Analysis Tool

To conduct mutation analysis on any kind of software artifacts, a mutation analysis tool that is able to generate mutants with selected mutation operators, execute tests against all the mutants, and compute and output the results of mutation analysis, is indispensable and critical.

For this research, an automated Android mutation analysis tool, called muDroid, was implemented. In addition, muDroid includes 17 new Android mutation operators designed in this research. This tool reuses the 15 traditional Java mutation operators [114] and the four deletion mutation operators [63, 67] in muJava, and extends the core Java mutation engine in muJava. Since Android apps have unique programming features, including the way they are developed, tested, distributed, and installed, this research implements muDroid with all the necessary features to address these unique aspects of Android apps. For example, muDroid is able to compile the source code and other necessary files to an APK file, install this APK file to an Android emulator or a mobile device, uninstall the old version of the Android app from an Android emulator or a mobile device, and control an unlimited number of Android emulators and mobile devices to execute in parallel. It is also compatible with tests developed with JUnit, and other major automated Android app testing
frameworks, such as Robotium [29] and Espresso [15]. In 2016, to improve the development experience for building Android apps, Android completely changed the methodology of developing Android apps by replacing Eclipse [14], Android Developer Tools (ADT) [5], and Apache ANT [12], with Android Studio [9] and Gradle [18]. This research also implements muDroid to be compatible with two versions of the Android development environment.

5.1.1 Functionality

Figure 4.1 illustrated the general process of mutation analysis on Android apps. muDroid completely facilitates the process of Android mutation testing. Following every step in the process, this section describes the functionality of muDroid.

The major features of muDroid are controlled through a command line for four reasons: (1) command line options provide a simple mechanism for automation, so that users can develop scripts to perform a batch of tasks, (2) command line options are available for external extensions, (3) because Android mutation testing requires a large amount of execution time, from several hours to a couple of days depending on the size of the app under test, thus command line options enable testers to conduct the tasks on cloud services or remote servers, (4) graphic user interfaces on mutation analysis tools may freeze during the execution, which would undermine the usability or introduce faults to the experiment.

Generating Mutants

Command: muDroidGen

The first step of performing mutation analysis is to select mutation operators and generate mutants. The command genmutes is designed to fulfill these tasks. Parameters of the command genmutes are listed as follows:

- `<operator name 1>  <operator name 2> ... <session name>`
- `project <project name> --package <package name>`

The user first identifies which mutation operators to use. The “<operator name>” is one of the 36 mutation operators (using the acronyms from Table 4.1) plus a simple
option to include all available mutation operators (ALL). muDroid allows users to choose any subset of the operators. The operator names can be either lower or upper case.

The “<session name>” takes the name of a specific test session. A test session defines an independent mutation testing unit, including source files, bytecode class files, test cases, mutants, and results.

- **--gradle**
  
  In 2016, Android released Android Studio [9] and updated the methodology for developing Android apps. Gradle [18] is the new build automation system that is used to manage the development of Android apps. The “--gradle” option is used to accommodate this switch. If this option is provided, the user indicates that the Android app under test is built with Gradle. Without this option, muDroid understands that the app under test is built with the old system, Apache ANT [12].

- **--debug**
  
  If the user would like to see more detailed intermediate output through the console, he or she can specify the “--debug” option in the command.

For example, Figure 5.1 shows an example command that generates Android mutants with TVD mutation operator for JustSit project. The command consists of several major parts:

- “java –jar muDroidGen.jar” launches the jar file

- “–TVD” specifies the use of the TVD mutation operator

- The first “JustSit” defines the session name

- “–project JustSit” gives the project name of the app under test, which might be different from the session name, particularly when the same app has multiple sessions for different sets of tests
• “–package com.brocktice.JustSit” defines the package name of the source code

Note that if a mutant cannot pass the Java compiler, it will be discarded. At the end of the process, the program displays the total number of the mutants generated. In this example, two TVD mutants are generated.

![Figure 5.1: An Example of Generating Mutants](image)

**Observing Mutants**

After generating mutants, muDroid provides users with a graphical user interface to observe every mutant, with the mutated code highlighted in colors. Figure 5.2 shows the two TVD mutants. The original version of the mutated file is in the left pane, while the mutated versions are in the right pane. The changes are highlighted in both panes. Several lines are used to connect the changes in two panes so that when the file grows larger, it is convenient for users to scroll the files to locate the changes and find out what the changes are. The first dropdown list includes all the sessions available in the muDroid home folder. After the user selects a session, the second dropdown list is automatically populated with the file names that have mutants. Once a file name is selected in the second dropdown list, a list of mutants is automatically shown in the scroll pane on the right. For example, two mutants, TVD_1 and TVD_2 are listed in the mutant list. Clicking on a mutant name will load and display the content, then compare with the original version.
Figure 5.2: Observing Mutants
Executing Tests on Mutants

One goal of Android mutation testing is to help testers design high quality tests. After generating mutants, users can design tests to kill them. The Android testing framework extends JUnit [21] to help testers design tests for their Android apps. Many users prefer to use external Android testing automation frameworks, such as Robotium [29]. muDroid enables both types of tests. Figure 5.3 shows an example test case written with Robotium. 

*Solo* is the main class for developing Robotium tests. It sends all types of different user actions to the app under test, such as *clickOnRadioButton* in the example. The test also includes test oracles to determine whether the program behavior is as expected.

```java
private Solo solo;
// Constructor
public test_ROR4() {
    super(TipsterActivity.class);
};
public void setUp() throws Exception {
    solo = new Solo(getInstrumentation(), getActivity());
};

// Test
public void testMethod() throws Exception {
    // check main activity
    assertEquals("TipsterActivity", getCurrentActivity());

    // click other
    solo.clickOnRadioButton(2);

    // enabled Other menu
    assertTrue(solo.getView(R.id.txtTipOther).isEnabled());
    solo.sleep(1000);

    // click 20% Button
    solo.clickOnRadioButton(1);

    // disabled Other menu
    assertFalse(solo.getView(R.id.txtTipOther).isEnabled());
}
```

Figure 5.3: An Example Test Case
Command: `muDroidRun`

The command `muDroidRun` is used to execute tests against mutants. Parameters of the command are listed as follows:

- `-adblocation <adb location> -session <session name>`
- `-file <file name1> -file <file name2> ...`
- `-test <test name1> -test <test name2> ...`

The option `-adblocation` specifies the location of Android Debug Bridge (ADB) file, which is a command line utility tool in Android SDK that can perform tasks on emulators and mobile devices, such as copying files, installing and uninstalling apps, and executing shell commands. The option `-file` lists the source files the user wants to test, and `-test` specifies the file names of tests. Both `-file` and `-test` accept an unlimited number of file names, so that the tester can run all or some of the tests on all or some of the files.

- `-runAll` (optional)

By default, to save execution cost, muDroid is set to an efficient mode such that once a test kills a mutant, the mutant is marked as killed and no further tests will execute against it. The `-runAll` option executes all the tests against all mutants, even if a mutant has been killed by another test. For example, if an app under test has 100 mutants, and the user designs 100 tests, without the `-runAll` option, there are 10,000 total iterations of execution, while with the `-runAll` option, the total number of iterations will be much fewer.

- `-device` (optional)

muDroid can control an unlimited number of mobile devices and emulators. However, the `-device` option tests a subset of connected devices by specifying their serial numbers.

- `-gradle` (optional)

Similar to the command for generating mutants, the `-gradle` option is used whenever
the app under test is built by Gradle.

Figure 5.4 shows an example command of executing tests against mutants. The command includes the following parts:

- “java –jar muDroidRun.jar” launches the jar file.
- “–adblocation” specifies the location of ADB file at C:/Android/android-sdk/platform-tools/adb.
- “–session” defines the session name as MunchLife.
- “–file” selects MunchLifeActivity as the file under test.
- “–runAll” enables the mode that executes all the tests on all the mutants.
- “–test” chooses the test with the name info.bpace.munchlife.test.test_ITR.

Figure 5.4: An Example Command of Executing Tests

Checking Results

After executing tests, muDroid saves the result of execution into a text (TXT) result file that lists the mutation score of the tests and which tests killed which mutants. The file name of the result includes a timestamp that indicates the exact finish date and time.

Figure 5.5 shows part of a result file. The mutation score is 77.39%. The mutant LOI_32 is killed by two tests, test_LOI3 and test(SDL)5. No tests killed mutant LOI_34 or SDL_91.
Note that the tests listed in the result file for each mutant may not be all the tests that could kill it, because the default execution mode is to not run killed mutants against subsequent tests. The --runAll option must be specified to obtain all killing tests for each mutant.

![Mutation Score: [0.7739]
LOI_32: [test_LOI3, test(SDL5]
ODL_30: [test_AOIS2, test_AOIU, test_AORB3, test_LOI3, test_ODL4, test(SDL5]
LOI_33: [test_LOI3]
LOI_34: []
LOI_35: [test_LOI3]
ODL_34: [test_ODL2]
SDL_91: []
ODL_33: [test_AOIS2, test_AOIU, test_ODL3]
SDL_92: [test(SDL1]
MDL_3: []
MDL_2: [test_AOIS2, test_AOIS3, test_BSW, test_LOI3, test_ROR, test_TVD]
LOI_66: [test_LOI2, test_AORB3]
LOI_67: [test_LOI3, test_AORB3]
MDL_4: [test_AOIS2, test_AOIS3, test_AOIU, test_AORB3, test(SDL5, test_TVD]
LOI_68: [test(SDL1, test_AORB3]
LOI_61: [test_AOIS2, test_AORB3, test_LOI3, test_ROR, test(SDL3, test(SDL4]
LOI_63: []
... ...](image)

Figure 5.5: An Example Partial Result File

5.1.2 Architecture of muDroid

Figure 5.6 shows the generate architecture of muDroid. It consists of two parsers (Java and XML), an information extractor, a mutant generator, a mutant observer, a test runner, a multithreading controller, and a result generator.

**Parsers and Information Extractor**

Since Android apps are primarily developed in Java and XML, two parsers are used by muDroid to recognize, understand, and mutate the source code.
For a Java parser, muDroid extends OpenJava, the parser component of muJava [114]. The Java parser component constructs a parser tree based on the Java source code of the Android app under test, and compiles mutated Java files to bytecode classes. In addition, muDroid implements an information extractor that can discover and store information used by mutation operators while parsing Java source code. For example, the OnClick Event Replacement mutation operator (ECR) requires the information of all the OnClick events designed in the Android app under test, so that it can replace an OnClick event with other similar events extracted from the app. This information extractor is critical to some Android mutation operators, such as ECR.
muDroid implements an XML parser to parse and mutate XML files, including both layout specification files and app configuration files. XML files are also parsed into a tree structure. The mutation is performed at the element level, that is, muDroid does not change the XML schema, or validate whether a given XML file is valid according to its schema or not. Unlike the parser component for Java, the XML parser does not compile XML files.

**Mutant Generator**

Unlike traditional Java programs, for which bytecode classes are interpreted and executed through Java Virtual Machines (JVM), Android apps must be compiled to APK files, which are uploaded and installed on mobile devices. Consequently, the mutant generator includes two sub-components: a log writer and an APK builder. After a selected mutation operator modifies the source code of the Android app under test, the APK builder collects all the files of the app, including Java bytecode classes, XML configuration and layout files, and graphics, then builds them to an APK file as a mutant of the Android app under test. If a mutant APK file cannot be compiled, the APK builder will discard it immediately. While building mutant APK files, the log writer assigns a number to every mutant, and keeps a log file that records how each mutant is changed, including its mutant number, the modified location (the method and the line number modified), the original, and the mutated code. Figure 5.7 shows part of a mutation log file. The file lists six AOIU mutants and six AOIS mutants. Take AOIU_1 as an example. The change is located at line 155, in method `void onCreate(android.os.Bundle)`, and a minus symbol is inserted in front of R.layout.main.

**Mutant Observer**

After generating mutants, users may want to check what these mutants are. Figure 5.2 showed a screenshot of the mutant observer. The mutant observer compares the original version of the Android app under test to the mutant selected from the mutant list, highlights the changes in blue, and connects the changes with lines for easy navigation.
Test Runner

The test runner first loads the original version of the Android app under test, randomly picks one connected Android emulator or real Android device, and installs the APK file of the app to it. If the selected device already has the app installed, which is very common in experimental studies, the test runner will replace the one installed on the target device, to ensure the execution results are from the original version of the Android app under test.

Then, the test runner executes all test cases on the original app and records the results. The results are compared to the results of each mutant to determine whether the mutant is killed. After that, the test runner picks every mutant and selects an available connected Android emulator or real Android device to execute tests on the mutant. Once the execution is finished, the test runner records the results from the mutant.

Depending on the option selected by the user, the test runner can stop the execution once a test kills the mutant, or can keep running with all the tests selected. The test runner is compatible with all the versions of the Android operating systems, and compatible with both Android emulators and real mobile devices.
Multithreading Controller

Computational cost is a major issue in Android mutation testing. Most Android devices and emulators run much slower than PCs or laptops, which results in a higher cost in terms of the execution time of Android mutation testing than conducting mutation testing on other programs. Consequently, muDroid implements a multithreading controller to parallelize the execution and shorten the overall time required to finish the Android mutation analysis.

Two pools are constructed before the test runner starts to execute tests on the Android app under test: one pool that contains all the connected Android emulators and real devices, and another pool that includes all the mutants generated. During the execution, the multithreading controller actively monitors the progress of every emulator and device. As soon as an emulator or device finishes execution, the multithreading controller selects an unexecuted mutant from the mutant pool, feeds it to the test runner, and notifies the test runner the serial ID of the available emulator or device. Then, the test runner installs the mutant, runs the tests, and records the results. Theoretically, muDroid can control an unlimited number of emulators and real devices.

Result Generator

The result generator is used to compare the results of the original version to the results of every mutant, determine whether the mutant is killed, compute an overall mutation score, and save all the results to a text file.

5.2 Empirical Evaluation of Android Mutation Testing

The first of three empirical evaluations tried to investigate the feasibility of applying mutation analysis to test Android apps. The motivation of this study was to verify whether Android mutation testing can be used to evaluate test cases designed with other testing criteria. I posed the following research questions:

- **RQ1:** Is it feasible to test real-world Android apps with mutation analysis?
• **RQ2:** How effective can test cases designed with traditional testing criteria be in killing mutants generated by Android mutation testing?

Statement coverage is one of the most frequently used testing coverage criteria in both industry and research. Many researchers use statement coverage to evaluate their techniques [115, 117, 124]. Just et al. [87] found that the correlation between mutation testing and real fault detection was statistically stronger than the correlation between statement coverage and real fault detection. Similarly, Android mutation testing is expected to be stronger than statement coverage. This empirical evaluation included five phases: selecting empirical subjects, designing test data with 100% statement coverage, generating mutants with muJava and Android mutation operators, executing tests against mutants, and analyzing results.

This study also checked whether results are consistent between the emulator and hardware devices, by using two Motorola MOTO G Android smartphones, one with the Dalvik Virtual Machine, and the other with ART, in addition to Android emulators. The smartphones were running in developer mode.

Note that when this study was conducted, only eleven Android mutation operators had been designed and implemented. The additional mutation operators were designed as a result of this study.

### 5.2.1 Empirical Subjects

This empirical study used eight Android apps as empirical subjects. These eight apps had been used in related research papers on Android testing [115, 117].

*Alarm Klock* [2] is an alarm clock app with advanced and customizable features. It has a 4.4 star rating by 6,366 reviews¹, and the Google Play store shows that the number of user installations is between 500,000 and 1,000,000.

*Jamendo for Android* [19] is an app for searching, streaming, and downloading free online music. It was obtained from F-Droid [16], a repository of free and open source

¹As of June, 2017
Android apps. It is not currently available on the Google Play store, so it does not have review or download data.

*JustSit* [22] is a timer app with an alarm used for meditation. Its latest version is 0.3.3, released in July 2010, with a 3.8 star rating from 140 users\(^2\).

*K-9 Mail* [23] is an email client app with a rich set of useful features that are not offered by similar email clients. On the Google Play store, it has a 4.2 star rating from 86,015 reviewers\(^3\), and several million user installations. Unlike other selected apps, which were developed by a small number of programmers, K-9 Mail is developed and released by an open source community with hundreds of contributors.

*MunchLife* [26] is a counter application for tracking levels achieved while playing the card game Munchkin. Its latest version is 1.4.4, released in February 2014, with a 4.3 star rating from 247 users\(^4\).

*PasswordMaker Pro* for Android [27] produces passwords for websites and other apps. It accepts a “master password” from the user, combines the URL or the name of the website requiring the password, and computes a unique password with hash algorithms. It has 23 classes in three different packages. On the Google Play store, the latest update was in January 2015, and it has a 3.8 star rating from 71 users\(^5\).

*TippyTipper* [33] calculates tips after taxes are added and splits bills among several customers. According to the Google Play store, the latest version 2.0 was released in December 2013 and has a 4.6 star rating from 795 users\(^6\). It was downloaded from its homepage. *TippyTipper* has five Activities: TippyTipper, SplitBill, Total, Settings, and About. It also has one Service: TipCalculatorService. Figure 5.8 illustrates three Activities: TippyTipper is on the left, SplitBill is in the middle, and Total is on the right.

*Tipster* [59] is similar to TippyTipper that is used to split payment and calculate tips. It is an example from Darwin’s book [60].

---

\(^2\)As of June, 2017  
\(^3\)As of June, 2017  
\(^4\)As of June, 2017  
\(^5\)As of June, 2017  
\(^6\)As of June, 2017
Table 5.1: Details of Empirical Subjects

<table>
<thead>
<tr>
<th>App</th>
<th>File</th>
<th>SLOC</th>
<th>ELOC</th>
<th>Lines of Dead Code</th>
<th>XML Elements</th>
</tr>
</thead>
</table>
| Alarm Klock | ActivityAlarmClock.java
             alarm_list.xml    | 290  | 127  | 3                  | 6            |
|             | AndroidManifest.xml   | 53   | 53   |                    | 35           |
| Jamendo     | HomeActivity.java
             main.xml          | 441  | 132  | 10                 | 10           |
|             | AndroidManifest.xml   | 146  | 146  |                    |              |
| JustSit     | JustSit.java
             main.xml          | 444  | 207  | 30                 | 13           |
|             | JsSettings.java       | 61   | 22   | 0                  | 6            |
|             | JsSettings.xml        | 52   | 52   |                    |              |
|             | AndroidManifest.xml   | 23   | 23   |                    | 14           |
| K-9 Mail    | ColorPickerDialog.java
                colorpicker_dialog.xml | 199  | 93   | 0                  | 7            |
|             | AndroidManifest.xml   | 214  | 214  |                    | 124          |
| MunchLife   | MunchLifeActivity.java
               main.xml          | 384  | 144  | 10                 | 12           |
|             | SettingsActivity.java
                preferences.xml    | 68   | 17   | 0                  | 5            |
|             | AndroidManifest.xml   | 32   | 32   |                    | 10           |
| PasswordMaker Pro | PasswordMakerPro.java
                        main.xml          | 606  | 343  | 26                 | 19           |
|             | AndroidManifest.xml   | 26   | 26   |                    | 13           |
| TippyTipper| TippyTipper.java
                main.xml          | 239  | 103  | 1                  | 20           |
|             | SplitBill.java        | 134  | 63   | 6                  | 31           |
|             | SplitBill.xml         | 93   | 93   |                    |              |
|             | Total.java            | 279  | 133  | 2                  |              |
|             | Total.xml             | 139  | 139  |                    | 44           |
|             | AndroidManifest.xml   | 32   | 32   |                    | 16           |
| Tipster     | TipsterActivity.java
                main.xml          | 297  | 115  | 0                  | 30           |
|             | AndroidManifest.xml   | 23   | 23   |                    | 7            |
| Total       | 5032                  | 3089 | 88   |                    | 515          |
This study used twelve classes along with their corresponding XML layout files, and the AndroidManifest.xml files from the eight apps. TippyTipper, MunchLifeActivity, JustSit, PasswordMakerPro, TipsterActivity, ActivityAlarmClock, and HomeActivity are the main Activity classes of their apps. Other classes were chosen based on their features in the corresponding apps. For example, in the TippyTipper app, the Activities SplitBill and Total were selected because they provide features including splitting and calculating tips and taxes, and generate a rich set of mutants. In addition, ColorPickerDialog of K-9 Mail is the only class that included event handlers for an OnTouch event. It was selected to ensure the ETR operator was evaluated.

Details about the empirical subjects are in Table 5.1. The Source Lines of Code (SLOC) and Executable Lines of Code (ELOC) for the Android classes were calculated by Emma [143], and the LOCs for XML files were counted within the Android IDE. An XML Document Object Model (DOM) parser was used to count the number of XML elements, because the number of elements is considered as a better way to measure size of XML files than the number of lines.

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

Table 5.1 also lists the number of lines of dead code manually identified for each class. The selected subjects had three types of dead code. First, if the default case is included in a switch-case block, but can never be reached with any user input, it is dead code. Second, if an event listener handles menu clicks, but no menu is on the screen, the entire listener class is dead code. Third, in a try-catch block, if it is impossible to throw and catch a required exception the entire catch block will be dead code.
5.2.2 Test Data Generation

First, an evolutionary algorithm-based Android test generation tool, EvoDroid [117], was used to create tests. It generated 744 test cases for the main Activity of *TippyTipper* through multiple generations. Ten tests were randomly chosen from the last generation, which covered 82% of the methods, 90% of the blocks, and 85% of the statements in the main Activity class, TippyTipper. After that, one additional test was designed by hand to achieve full statement coverage.

For the nine other Android classes and their associated XML layout files, test inputs were manually designed to achieve 100% statement coverage (Table 5.1), excluding the dead code. All available test sets designed for each app were executed against APD mutants of AndroidManifest.xml files. For example, the test set for AndroidManifest.xml of *TippyTipper* consists of all the test cases designed to test the Activities of TippyTipper, SplitBill, and Total.

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

5.2.3 Mutant Generation

<table>
<thead>
<tr>
<th>App</th>
<th>Component (Java and XML Layout)</th>
<th>muJava Mutants</th>
<th>Android Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm Klock</td>
<td>ActivityAlarmClock</td>
<td>161</td>
<td>235</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td>6</td>
</tr>
<tr>
<td>Jamendo</td>
<td>HomeActivity</td>
<td>237</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td>7</td>
</tr>
<tr>
<td>JustSit</td>
<td>JustSit</td>
<td>415</td>
<td>241</td>
</tr>
<tr>
<td></td>
<td>JsSettings</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>K-9 Mail</td>
<td>ColorPickerDialog</td>
<td>551</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td>17</td>
</tr>
<tr>
<td>MunchLife</td>
<td>MunchLifeActivity</td>
<td>534</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>SettingsActivity</td>
<td>47</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td>1</td>
</tr>
<tr>
<td>PasswordMakerPro</td>
<td>PasswordMakerPro</td>
<td>515</td>
<td>379</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>TippyTipper</td>
<td>105</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>SplitBill</td>
<td>124</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>231</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td>4</td>
</tr>
<tr>
<td>Tipster</td>
<td>TipsterActivity</td>
<td>327</td>
<td>118</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>3275</strong></td>
<td><strong>1706</strong></td>
</tr>
</tbody>
</table>

19 Java method-level mutation operators from muJava [114], and 11 Android mutation operators were used to generate mutants. Then, these mutants were compiled into installable APK files. Depending on the size and content of the subjects, generating a mutant and compiling it as an APK file took up to ten seconds on a MacBook Pro with a 2.6 GHz Intel i7 processor and 16 GB memory.
Table 5.2 lists the results of mutants generation. muDroid generated a total of 3275 mutants from the 19 method-level operators. The number of muJava mutants ranged from 28 (in JsSettings of JustSit) to 551 (in ColorPickerDialog of K-9 Mail). The 11 new Android mutation operators generated 1706 valid Android mutants for 12 Android classes along with their corresponding XML layout files, and 5 AndroidManifest.xml files (TippyTipper, MunchLife, K-9 Mail, Alarm Klock, and Jamendo). The number of Android Java mutants ranged from 7 (in SettingsActivity of MunchLife) to 379 (in PasswordMakerPro of PasswordMakerPro), and Android XML mutants ranged from one to 17.

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

Each Android app has an AndroidManifest.xml file, but three AndroidManifest.xml files (in subjects JustSit, PasswordMakerPro, and Tipster) did not have any mutants. Thus, they are not listed in Table 5.2.

5.2.4 Empirical Results and Discussion

After generating all the mutants, muDroid loaded and executed 100% statement coverage test sets against these mutants. Table 5.3 summarizes results from running both muJava and Android mutants. Equivalent mutants were removed by hand.

Table 5.4 shows results for each mutation operator. The first group contains results from the muJava traditional mutants, and the second group presents results from the Android
<table>
<thead>
<tr>
<th>App</th>
<th>File</th>
<th>muJava Mutants</th>
<th>Android Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Killed</td>
<td>Equiv.</td>
</tr>
<tr>
<td>Alarm Klock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ActivityAlarmClock</td>
<td>161</td>
<td>114</td>
<td>12</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jamendo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HomeActivity</td>
<td>237</td>
<td>171</td>
<td>13</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JustSit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JustSit</td>
<td>415</td>
<td>153</td>
<td>50</td>
</tr>
<tr>
<td>JsSettings</td>
<td>28</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>K-9 Mail</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ColorPickerDialog</td>
<td>551</td>
<td>271</td>
<td>56</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MunchLife</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MunchLifeActivity</td>
<td>534</td>
<td>324</td>
<td>72</td>
</tr>
<tr>
<td>SettingsActivity</td>
<td>47</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PasswordMakerPro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PasswordMakerPro</td>
<td>515</td>
<td>229</td>
<td>89</td>
</tr>
<tr>
<td>TippyTipper</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TippyTipper</td>
<td>105</td>
<td>71</td>
<td>4</td>
</tr>
<tr>
<td>SplitBill</td>
<td>124</td>
<td>52</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>231</td>
<td>123</td>
<td>29</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>n/a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tipster</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TipsterActivity</td>
<td>327</td>
<td>234</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>3275</td>
<td>1778</td>
<td>377</td>
</tr>
<tr>
<td>Median</td>
<td>234</td>
<td>138</td>
<td>28</td>
</tr>
<tr>
<td>Mean</td>
<td>272.9</td>
<td>148.2</td>
<td>31.4</td>
</tr>
</tbody>
</table>
mutants.

**RQ1: Is it feasible to test real-world Android apps with mutation analysis?**

Table 5.3 shows that muDroid successfully generated 3,275 Java traditional method-level mutants, and 1,706 Android mutants for 8 real-world open source Android apps, and executed pre-designed 100% statement coverage test sets on these mutants. Across all subjects, 1,778 of 3,275 muJava mutants and 530 of 1,706 Android mutants were killed by 100% statement coverage test sets, after eliminating equivalent mutants by hand analysis.

In addition, to assess whether the emulator had any effect on the evaluation, this study executed the tests on different smartphones using Dalvik and ART. The mutation scores were identical in all the environments. Therefore, the Android Virtual Machines used does not have any effect on the empirical study. However, the emulator was much slower than real devices, even with the Intel Hardware Accelerated Execution Manager (HAXM) installed.

**RQ2: How effective can test cases designed with traditional testing criteria be in killing mutants generated by Android mutation testing?**

The premise of mutation testing is that if a software program has a fault, there usually are some mutants that can only be killed by a test that also detects that fault. Therefore, effective tests should be able to kill as many mutants as they can, because each mutant can be considered as a software fault. The mutation score of a test set is the percentage of non-equivalent mutants killed by this test set. It ranges from 0% to 100% and directly indicates whether a given test set is effective or not.

Table 5.3 shows the mutation scores (MS columns) of the 100% statement coverage test sets designed for the subjects. These mutation scores are computed after removing equivalent mutants. In other words, the percentages show how many mutants are killed relative to how many can be killed.

The mutation scores for the muJava mutants ranged from 0.419 (in JustSit of *JustSit*) to 0.78 (in TipsterActivity of *Tipster*), with a mean of 0.622 and a median of 0.644. For Android mutants, the mutation scores ranged from 0.455 (in SplitBill of *TippyTipper*) to 0.885 (in HomeActivity of *Jamendo for Android*), with a mean of 0.666 and a median of
Table 5.4: Empirical Results for Each Mutation Operator

<table>
<thead>
<tr>
<th>Operator</th>
<th>Killed Mutants</th>
<th>Equivalent Mutants</th>
<th>Live Mutants</th>
<th>Total Mutants</th>
<th>Mutation Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional Mutants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AODU</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0.750</td>
</tr>
<tr>
<td>AOIS</td>
<td>249</td>
<td>151</td>
<td>120</td>
<td>520</td>
<td>0.675</td>
</tr>
<tr>
<td>AOIU</td>
<td>253</td>
<td>17</td>
<td>112</td>
<td>382</td>
<td>0.693</td>
</tr>
<tr>
<td>AORB</td>
<td>113</td>
<td>4</td>
<td>71</td>
<td>188</td>
<td>0.614</td>
</tr>
<tr>
<td>AORS</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
</tr>
<tr>
<td>CDL</td>
<td>22</td>
<td>9</td>
<td>18</td>
<td>49</td>
<td>0.550</td>
</tr>
<tr>
<td>COD</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0.857</td>
</tr>
<tr>
<td>COI</td>
<td>70</td>
<td>4</td>
<td>55</td>
<td>129</td>
<td>0.560</td>
</tr>
<tr>
<td>COR</td>
<td>18</td>
<td>0</td>
<td>14</td>
<td>32</td>
<td>0.563</td>
</tr>
<tr>
<td>LOI</td>
<td>296</td>
<td>7</td>
<td>118</td>
<td>421</td>
<td>0.715</td>
</tr>
<tr>
<td>LOR</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0.000</td>
</tr>
<tr>
<td>ODL</td>
<td>82</td>
<td>22</td>
<td>84</td>
<td>188</td>
<td>0.494</td>
</tr>
<tr>
<td>ROR</td>
<td>169</td>
<td>51</td>
<td>186</td>
<td>406</td>
<td>0.476</td>
</tr>
<tr>
<td>SDL</td>
<td>471</td>
<td>109</td>
<td>309</td>
<td>889</td>
<td>0.604</td>
</tr>
<tr>
<td>VDL</td>
<td>26</td>
<td>3</td>
<td>26</td>
<td>55</td>
<td>0.500</td>
</tr>
<tr>
<td>Subtotal</td>
<td>1778</td>
<td>377</td>
<td>1120</td>
<td>3275</td>
<td>0.614</td>
</tr>
<tr>
<td>Android Mutants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APD</td>
<td>10</td>
<td>4</td>
<td>21</td>
<td>35</td>
<td>0.323</td>
</tr>
<tr>
<td>BWD</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>1.000</td>
</tr>
<tr>
<td>BWS</td>
<td>0</td>
<td>0</td>
<td>99</td>
<td>99</td>
<td>0.000</td>
</tr>
<tr>
<td>ECR</td>
<td>111</td>
<td>0</td>
<td>4</td>
<td>115</td>
<td>0.965</td>
</tr>
<tr>
<td>ETR</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1.000</td>
</tr>
<tr>
<td>FON</td>
<td>146</td>
<td>949</td>
<td>25</td>
<td>1120</td>
<td>0.854</td>
</tr>
<tr>
<td>IPR</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>1.000</td>
</tr>
<tr>
<td>ITR</td>
<td>181</td>
<td>0</td>
<td>29</td>
<td>210</td>
<td>0.862</td>
</tr>
<tr>
<td>MDL</td>
<td>18</td>
<td>1</td>
<td>5</td>
<td>24</td>
<td>0.783</td>
</tr>
<tr>
<td>ORL</td>
<td>13</td>
<td>0</td>
<td>35</td>
<td>48</td>
<td>0.271</td>
</tr>
<tr>
<td>TWD</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>0.600</td>
</tr>
<tr>
<td>Subtotal</td>
<td>530</td>
<td>954</td>
<td>222</td>
<td>1706</td>
<td>0.705</td>
</tr>
<tr>
<td>Total</td>
<td>2308</td>
<td>1331</td>
<td>1342</td>
<td>4981</td>
<td>0.632</td>
</tr>
</tbody>
</table>
0.674, excluding the three AndroidManifest.xml files. Obviously, by missing around 39% of Java traditional method-level mutants, and 30% of Android mutants, the 100% statement coverage tests used in this empirical study were not very effective at killing mutants.

**Discussion**

Arithmetic Operator Replacement (AORS) and Logical Operator Replacement (LOR) mutants have the lowest mutation scores of 0, meaning that none were killed by the statement coverage test sets. These two operators only generated five mutants, so this low percentage probably is not meaningful.

Among the Android mutation operators, none of the Button Widget Switch (BWS) mutants were killed by any test, as the statement coverage test sets could not ensure the locations (either relative, or absolute) of button widgets.

The highest mutation score among the traditional muJava mutants were for Conditional Operator Deletion (COD), 0.857. All of the Android mutants for OnTouch Event Replacement (ETR), Intent Payload Replacement (IPR), and Button Widget Deletion (BWD) were killed.

The APD operator (permission deletion) only applies to AndroidManifest.xml files. The principle of least privilege [144] requires that an app should only request necessary permissions from the Android system. If an app still works correctly after APD deletes its permissions (that is, the mutant is equivalent), the permission was unnecessary and granting it could create a security or privacy threat.

In this empirical study, none of the four APD mutants of *TippyTipper* were killed. Since the 100% statement coverage test sets were only designed to cover three out of five Activities in the app, testing could not show whether those Activities needed the permissions. To verify whether the permissions were necessary to the app, a detailed hand analysis of the needs of all the Activities were conducted, which found that none of the Activities used any of the four permissions requested (WRITE_SETTINGS, WAKE_LOCK, MODIFY_AUDIO_SETTINGS, and VIBRATE). Therefore, it turned out that *TippyTipper* does
not need any of them. These four APD mutants were actually equivalent mutants. Additionally, fourteen live APD mutants of _K-9 Mail_ were judged not equivalent after manual analysis.

In Table 5.4, 949 of 1,120 (84.7%) FON mutants are equivalent, which is the highest among all mutation operators. This is because many objects in Android apps can never be null or empty, which makes the “fail on null” statement impossible to trigger. Figure 5.5 shows an example equivalent FON mutant. Since object _array_ is newly created and correctly initialized, no test can trigger the “fail on null” statement. However, when FON generates mutants, muDroid cannot decide whether an object can be null or empty. Even though manually identifying and filtering these equivalent FON mutants is straightforward and not time-consuming, it is highly recommended that an improvement in the implementation of muDroid should be carried out to avoid generating these equivalent mutants, to reduce the cost of Android mutation testing.

<table>
<thead>
<tr>
<th>Original</th>
<th>FON Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ArrayList&lt;String&gt; array = new ArrayList&lt;&gt;();</code></td>
<td><code>ArrayList&lt;String&gt; array = new ArrayList&lt;&gt;();</code></td>
</tr>
<tr>
<td><code>array.add(&quot;test&quot;);</code></td>
<td><code>FailOnNull (array);</code></td>
</tr>
<tr>
<td><code>array.add(&quot;test&quot;);</code></td>
<td><code>array.add(&quot;test&quot;);</code></td>
</tr>
</tbody>
</table>

### 5.2.5 Threats to Validity

This empirical evaluation has several threats to validity, which could have influenced the experimental results. However, this study considered these threats and took measures to minimize and avoid them.

**Internal validity:** Dead code in the subjects and equivalent mutants were identified manually by one person. In addition, all the computation and analysis in this empirical study were conducted using Microsoft Excel. Some tests used in this study were first generated by _evoDroid_, an evolutionary algorithm based Android test generation tool, then
augmented by hand to achieve 100% statement coverage. Other tests were manually created without the support of any tool. Human mistakes become a potential threat to this study.

**External validity:** Like most software engineering experiments, it is not possible to guarantee the representativeness of selected subjects. The subjects were selected to have different sizes, from different sources, and they were in various domains. Also, the fact that all the subjects were used by previous researchers provide consistency across multiple studies. This research is designed for native Android apps that are applications traditionally developed for running on the Android operating systems with the features supported by Android SDK libraries. Hybrid apps and HTML\(\text{5}\) apps were not used in this empirical study, because these apps are implemented with elements of web applications that are not based on the Android platform. The results may differ when testing Hybrid apps and HTML\(\text{5}\) apps.

**Construct validity:** The implementation of Android mutation operators and the Android mutation testing tool, muDroid, may include software faults. They were constantly tested during the implementation process, to ensure they work as expected. Therefore, in this empirical study they were assumed to work correctly.

### 5.3 Experimental Evaluation of Fault Detection Effectiveness

The previous empirical study took the first steps toward applying mutation testing to Android apps and its results show that it is feasible to test real-world Android apps with mutation analysis. The next experimental evaluation focused on evaluating the fault detection effectiveness of Android mutation testing and comparing with other Android testing techniques. The research used two sets of software faults: naturally occurring faults mined from the subject apps’ source code repositories and crowdsourced faults collected from experienced Android developers.

This experimental evaluation addresses the following three research questions:
• **RQ3:** How effective is Android mutation analysis at testing Android apps? Specifically, how many faults can be detected by mutation-generated tests?

• **RQ4:** 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?

• **RQ5:** Is there any difference between using naturally occurring faults and using crowdsourced faults in empirical evaluations?

To better demonstrate the fault detection effectiveness of different techniques, the null and alternative hypotheses (Hypotheses\textsubscript{A}) for naturally occurring faults are:

**Null hypothesis** ($H_0$): There is no significant difference between the numbers of naturally occurring faults detected by Android mutation testing and other Android testing techniques.

**Alternative hypothesis** ($H_1$): There is a significant difference between the numbers of naturally occurring faults detected by Android mutation testing and other Android testing techniques.

The null and alternative hypotheses (Hypotheses\textsubscript{B}) for crowdsourced faults are:

**Null hypothesis** ($H_0$): There is no significant difference between the numbers of crowdsourced faults detected by Android mutation testing and other Android testing techniques.

**Alternative hypothesis** ($H_1$): There is a significant difference between the numbers of crowdsourced faults detected by Android mutation testing and other Android testing techniques.

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. The four other techniques represented the current state-of-the-art and state-of-the-practice, and no structural-based techniques or tools for testing mobile apps were available.
to compare with.

5.3.1 Experimental Subjects

Nine subject apps were used in this experiment. Six were re-used from the previous experiment, including Alarm Klock [2], Jamendo [19], JustSit [22], MunchLife [26], TippyTipper [33], and Tipster [59]. In addition, three new subject apps were selected, including AndroidomaticKeyer [11], Lolcat Builder [24], and WorldClock [35]. AndroidomaticKeyer [11] translates user-entered text into Morse code and plays it through Android devices’ sound output. It is the only subject app that accesses the GPS sensor to process location data. It has a 4.1 star user rating. Lolcat Builder [24] lets users edit photos by adding captions and text, then save or share them. It has a 3.8 star rating from 376 users. WorldClock [35] lets users search and add different locations in the world, then gives the current time at each selected place. It has a 4.1 star rating.

Table 5.6 summarizes the nine experiment subjects. The information regarding Lines of Code (LOC) in the Java files was obtained by using Metrics [25], a plugin tool for Eclipse. As an XML file can be formatted into varied forms with different numbers of lines, the number of XML elements (nodes) in XML files is a better and more reliable indicator to measure than Lines of Code (LOC). Therefore, in this experimental evaluation, an XML parser program was developed to count the numbers of XML nodes in the XML layout and manifest (configuration) files.

According to Table 5.6, PlayerActivity.java of Jamendo is the largest Java file among all nine subject apps, with 662 LOC. Its XML layout file, player.xml, is also the largest XML file, with 45 XML elements. The smallest Java file is Help.java of AndroidomaticKeyer, which has 10 LOC. The smallest XML file is pending_alarms_item.xml of AlarmKlock with only one XML element.

---

7 As of June, 2017
8 As of June, 2017
9 As of June, 2017
<table>
<thead>
<tr>
<th>App</th>
<th>File</th>
<th>LOC</th>
<th>XML Elements</th>
<th>muJava Mutants</th>
<th>Android Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>ActivityAlarmClock.java</td>
<td>227</td>
<td>6</td>
<td>161</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>alarm_list.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>alarm_list_item.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ActivityAlarmNotification.java</td>
<td>175</td>
<td>10</td>
<td>210</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>notification.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ActivityAlarmSettings.java</td>
<td>395</td>
<td>3</td>
<td>401</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>settings_item.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>settings.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ActivityAppSettings.java</td>
<td>81</td>
<td></td>
<td>39</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>35</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>AlarmClockService.java</td>
<td>240</td>
<td></td>
<td>388</td>
<td>210</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td>AndroidomaticKeyerActivity.java</td>
<td>567</td>
<td>14</td>
<td>1001</td>
<td>408</td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>StraightKeyActivity.java</td>
<td>88</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>sk.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Help.java</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>help.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GeoHelper.java</td>
<td>196</td>
<td></td>
<td>507</td>
<td>447</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>21</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Jamendo</td>
<td>AlbumActivity.java</td>
<td>212</td>
<td>11</td>
<td>263</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>album.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ArtistActivity.java</td>
<td>142</td>
<td>8</td>
<td>81</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>artist.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DownloadActivity.java</td>
<td>188</td>
<td>14</td>
<td>206</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>download.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HomeActivity.java</td>
<td>304</td>
<td>10</td>
<td>324</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PlayerActivity.java</td>
<td>662</td>
<td>45</td>
<td>663</td>
<td>551</td>
</tr>
<tr>
<td></td>
<td>player.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PlaylistActivity.java</td>
<td>176</td>
<td>7</td>
<td>210</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>playlist.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RadioActivity.java</td>
<td>242</td>
<td>11</td>
<td>196</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>search.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SettingsActivity.java</td>
<td>36</td>
<td>9</td>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>settings.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>downloadService.java</td>
<td>90</td>
<td></td>
<td>68</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>PlayerService.java</td>
<td>220</td>
<td></td>
<td>232</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>21</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>JsSettings.java</td>
<td>36</td>
<td>6</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>
Table 5.6: Details of Experimental Subjects

<table>
<thead>
<tr>
<th>App</th>
<th>File</th>
<th>LOC</th>
<th>XML Elements</th>
<th>muJava Mutants</th>
<th>Android Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>JustSit</td>
<td>RunTimer.java</td>
<td>67</td>
<td>3</td>
<td>131</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>run_timer.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JustSit.java</td>
<td>351</td>
<td>13</td>
<td>394</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JsAbout.java</td>
<td>23</td>
<td>6</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>about.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>14</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Lolcat Builder</td>
<td>LolcatActivity.java</td>
<td>482</td>
<td>8</td>
<td>581</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>lolcat_activity.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>12</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MunchLife</td>
<td>MunchLifeActivity.java</td>
<td>285</td>
<td>12</td>
<td>534</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SettingsActivity.java</td>
<td>47</td>
<td>5</td>
<td>47</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>preferences.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>10</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TippyTipper</td>
<td>Total.java</td>
<td>218</td>
<td>44</td>
<td>231</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>Total.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TippyTipper.java</td>
<td>179</td>
<td>20</td>
<td>105</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SplitBill.java</td>
<td>108</td>
<td>31</td>
<td>124</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>SplitBill.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Settings.java</td>
<td>51</td>
<td>12</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>settings.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>About.java</td>
<td>26</td>
<td>10</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>about.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TipCalculatorService.java</td>
<td>244</td>
<td></td>
<td>1102</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>12</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tipster</td>
<td>TipsterActivity.java</td>
<td>180</td>
<td>30</td>
<td>327</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>7</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>WorldClock</td>
<td>WorldClockActivity.java</td>
<td>147</td>
<td>3</td>
<td>193</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>worldclock_main.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>timezoneedit.java</td>
<td>129</td>
<td>7</td>
<td>113</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>timezone_edit.xml</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>15</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>6,879</strong></td>
<td><strong>530</strong></td>
<td><strong>8,940</strong></td>
<td><strong>4,739</strong></td>
<td></td>
</tr>
</tbody>
</table>

19 Java traditional method-level mutation operators defined in muJava [114] yielded 8,940.
muJava mutants. 17 Android mutation operators yielded 4,739 Android mutants for Activities, Services, XML layout files, and configuration files of the subjects. The number of muJava mutants ranges from four (Help.java of AndroidomaticKeyer and About.java of TippyTipper) to 1,102 (TipCalculatorService.java of TippyTipper). The AndroidManifest.xml files in two subjects (TippyTipper and Tipster) did not generate any Android mutants. PlayerActivity.java and its associated XML layout file, player.xml, together generated 551 Android mutants, which is the highest number of Android mutants.

5.3.2 Experimental Procedure

The experimental procedure included five major steps to evaluate the fault detection effectiveness of Android mutation testing:

1. **Generate mutants:** 19 Java traditional method-level mutation operators and 17 Android mutation operators were used to generate mutants for the 9 experimental subjects.

2. **Design tests:** Tests were manually designed to specifically kill all non-equivalent mutants. All the tests were created with the support of JUnit [21] and Robotium [29]. Equivalent mutants were identified by hand during this step and eliminated from the experiment.

3. **Collect faults:** Two types of software program faults were collected in this experiment: naturally occurring faults and crowdsourced faults. Sections 5.3.3 and 5.3.4 describe this step. Each faulty version of the experimental subject apps was saved as a separate Android app project, so that they can be simply organized and deployed.

4. **Detect faults:** These mutation-adequate tests were executed against each faulty version of the experimental subject apps. Then, the results of whether the faults were detected or not were recorded.

5. **Compute results:** The fault detection effectiveness of Android mutation testing was computed for naturally occurring faults and crowdsourced faults.
After that, two additional steps were conducted to evaluate the fault detection effectiveness of the four other Android testing tools.

1. **Fault detection:** Each Android testing tool was used to test every faulty version of the experimental subject apps. Their fault detection results were recorded.

2. **Results comparison:** The fault detection effectiveness of the four Android testing techniques for naturally occurring and hand-seeded Android faults was calculated, evaluated, and compared.

Note that each faulty version of the experimental subject apps was deployed and tested on a clean Android emulator or smartphone to ensure each execution was independent from the others.

### 5.3.3 Collecting Naturally Occurring Faults

This section discusses how the naturally occurring faults were collected. As all the nine subject apps are open source Android apps, each one has its own GitHub repository. Usually, when developers fix a software fault of open source apps, they submit a new commit to the repository and describe what, why, and where the change is. An issue tracking system in the repository can be used to retrieve the information about the faults and make it possible to reproduce them. Thus, in this experiment, naturally occurring faults were collected by searching the open source repository of each experimental subject app.

Using the keywords *fix, fault, bug, issue,* and *incorrect,* every commit in the nine GitHub source code repositories was examined. In each commit related to a fault, three types of data were collected: the source code of the faulty version (the prior version), the source code of the fixed version, and the description of the commit. Additionally, some developers linked their commits with the bug reports in their app’s issue tracking system to further describe the issues that were fixed.

Many commits contained the keywords but actually did not fix a software fault. For example, several commits were labeled with the keyword “fix,” even though they did not fix
a fault, so the detail of every change had to be carefully examined. Other comments were labeled with “issue,” even though they just changed usability or performance. Commits that were used to fix typos in message strings or comments were not considered valid, either. In summary, non-fault fixing commits had to be discarded by manual examination before proceeding to the next experimental step.

After discarding those non-fault fixing commits, every collected naturally occurring fault was verified 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 Android 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. Faults that could not be reproduced in our experimental environment were discarded.

Table 5.7 summarizes the naturally occurring faults collected in this experimental step. Two subjects, Tipster and Lolcat Builder, do not have commits explicitly related to fixed faults. AlarmKlock has 13 collected faults, in which 7 are reproducible. Both are the highest among all the subjects. Jamendo also has 13 collected faults, but only 6 of them can be reproduced. Both JustSit and WorldClock have only one reproducible fault. In total, 51 commits with fault fixing activities were collected, in which 25 faults could be reproduced in our experimental environment, and 26 faults were discarded.

5.3.4 Collecting Crowdsourced Faults

This section describes how crowdsourced faults were collected. Since the number of naturally occurring faults collected in this experiment was too low to conduct an empirical study and draw any reasonable conclusion, additional faults were required. This study used crowdsourcing to create additional faults. Anonymous Android developers were recruited from a crowdsourcing website\textsuperscript{10} to hand-seed faults into our nine experiment subject apps.

\textsuperscript{10}www.Freelancer.com
Table 5.7: Numbers of Naturally Occurring Faults Collected for Each Subject App

<table>
<thead>
<tr>
<th>Apps</th>
<th>Total Collected Commits</th>
<th>Discarded Commits</th>
<th>Commits Kept as Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>13</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td>8</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Jamendo</td>
<td>13</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>JustSit</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MunchLife</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>WorldClock</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>51</strong></td>
<td><strong>26</strong></td>
<td><strong>25</strong></td>
</tr>
</tbody>
</table>

After submitting the recruiting post on the crowdsourcing website, more than 30 Android developers from different countries proposed to participate in this study. To ensure the candidates had the needed expertise, every applicant’s prior project history on the crowdsourcing website was carefully reviewed. An ideal candidate must have established records in Android development, have finished previous projects on time and on budget with a 5-star rating, and be able to communicate in English. Their identities were also verified by phone, email, or Facebook. After this initial verification, each applicant was interviewed to determine his or her knowledge and experience in developing and testing Android apps.

Even though all the participants were experienced Android developers, they were not familiar with seeding faults into software programs. For example, one freelancer kept trying to identify possible faults in the experimental subjects. After several rounds of explanation and discussion with him, he still could not understand his task was to hand-seed software faults, not finding faults. Thus, he was eventually replaced by another candidate.

It is worth mentioning that only those applicants who understood the project in general but **did not** know mutation testing were selected. They were not provided any instructions or guidelines on how to seed faults. The participants were allowed to seed any faults based on their own developing and testing experience.

After this rigorous process, our crowd was composed of seven freelancers from India, Bangladesh, Pakistan, and Ukraine. As incentive, each valid seeded fault was awarded $1,
and they were allowed to seed up to 200 faults in the nine experiment subject apps.

Every *microtask*, that is hand-seeding a software fault, was considered to be valid only if:

1. It changed the source code, including Java classes, XML files, or any other files in the project.

2. The changed app could be compiled and executed in our experimental 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.

5. This incorrect behavior must be observable.

Table 5.8 lists the numbers of hand-seeded faults created by the participants of this study. The number of faults ranged from 49 from freelancer F5 to 148 from F4. The number of faults per subject ranged from 25 to 95. Only one freelancer seeded faults into AndroidomaticKeyer. Overall, this study collected 589 crowdsourced faults for the nine subject apps.

**Table 5.8: Numbers of Hand-seeded Faults for Each App before Removing Mutants**

<table>
<thead>
<tr>
<th>Apps</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>6</td>
<td>14</td>
<td>6</td>
<td>16</td>
<td>9</td>
<td>18</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Jamendo</td>
<td>15</td>
<td>3</td>
<td>11</td>
<td>28</td>
<td></td>
<td></td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>JustSit</td>
<td>15</td>
<td>7</td>
<td>15</td>
<td>35</td>
<td>15</td>
<td>8</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>Lolcat Builder</td>
<td>12</td>
<td>14</td>
<td>34</td>
<td>13</td>
<td>5</td>
<td></td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>MunchLife</td>
<td>12</td>
<td>8</td>
<td>10</td>
<td>24</td>
<td>17</td>
<td>8</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>TippyTipper</td>
<td>25</td>
<td>10</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Tipster</td>
<td>5</td>
<td>25</td>
<td>20</td>
<td>25</td>
<td>5</td>
<td>5</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>WorldClock</td>
<td>3</td>
<td></td>
<td>27</td>
<td>11</td>
<td>9</td>
<td></td>
<td>41</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>68</td>
<td>96</td>
<td>61</td>
<td>148</td>
<td>49</td>
<td>98</td>
<td>78</td>
<td>589</td>
</tr>
</tbody>
</table>
Table 5.9: Numbers of Hand-seeded Faults for Each App after Removing Mutants

<table>
<thead>
<tr>
<th>Apps</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>% Non-mutant Faults</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>5</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>8</td>
<td>16</td>
<td></td>
<td>85.51%</td>
<td>59</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>100.00%</td>
<td>25</td>
</tr>
<tr>
<td>Jamendo</td>
<td>11</td>
<td>3</td>
<td>6</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>70.18%</td>
<td>40</td>
</tr>
<tr>
<td>JustSit</td>
<td>14</td>
<td>7</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>7</td>
<td></td>
<td>62.11%</td>
<td>59</td>
</tr>
<tr>
<td>Lolcat Builder</td>
<td>9</td>
<td>14</td>
<td>18</td>
<td>11</td>
<td>4</td>
<td></td>
<td></td>
<td>71.79%</td>
<td>56</td>
</tr>
<tr>
<td>MunchLife</td>
<td>11</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>7</td>
<td>8</td>
<td></td>
<td>65.82%</td>
<td>52</td>
</tr>
<tr>
<td>TippyTipper</td>
<td></td>
<td>22</td>
<td>10</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>78.33%</td>
<td>47</td>
</tr>
<tr>
<td>Tipster</td>
<td>3</td>
<td>22</td>
<td>17</td>
<td>14</td>
<td>4</td>
<td>4</td>
<td></td>
<td>75.29%</td>
<td>64</td>
</tr>
<tr>
<td>WorldClock</td>
<td>3</td>
<td></td>
<td>15</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td>85.37%</td>
<td>35</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>56</td>
<td>85</td>
<td>53</td>
<td>78</td>
<td>26</td>
<td>67</td>
<td>72</td>
<td></td>
<td>437</td>
</tr>
<tr>
<td>% of Non-mutant Faults</td>
<td>82.35%</td>
<td>88.54%</td>
<td>86.89%</td>
<td>52.70%</td>
<td>53.06%</td>
<td>68.37%</td>
<td>92.31%</td>
<td>74.19%</td>
<td></td>
</tr>
</tbody>
</table>
One problem with not giving detailed guidelines about the type of faults needed is that the participants quite naturally created faults that were actually the same as mutants generated in this study. Thus, each crowdsourced fault created had to be hand-inspected, and faults that were also mutants were eliminated to avoid biasing the results in favor of mutation. Table 5.9 lists the numbers of hand-seeded faults after excluding mutants. All 25 hand-seeded faults for AndroidomaticKeyer were non-mutants, while JustSit only had 59 non-mutant faults. Figure 5.9 compares the numbers of faults before and after removing mutant-faults. Grey bars indicate the numbers of faults before removing mutant-faults for each participant, and white bars represent the numbers after the removal. One freelancer (F4) had more than 47% mutant-faults. Freelancer F7 had the least, less than 8%. After removing these mutant-faults, a total of 437 non-mutant crowdsourced faults were available to use in this experiment.
5.3.5 Other Android App Testing Techniques

This experiment evaluated the fault detection effectiveness of four Android app testing tools. The same faults were used. Since at the time of conducting the experiment, no other code-based Android testing tools existed, a direct comparison with Android mutation testing is difficult. However, these four Android testing tools represented both the state-of-the-art and the state-of-the-practice. This section introduces these four Android testing techniques.

Monkey [8] is officially provided by the Android SDK, and is widely used by Android developers. It can run on both emulators and real devices. Monkey sends pseudo-random user events to the app, such as touching, flipping, and zooming. Many developers use Monkey to stress-test their apps.

Dynodroid [115] is an open source project developed by the Android testing research community. Dynodroid has been used by other researchers to evaluate the effectiveness of Android testing techniques. 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. Unlike Monkey, Dynodroid considers system events in addition to UI events to generate both human and machine inputs.

PUMA [81] is a dynamic analysis tool that enables scalable UI automation for Android apps. It incorporates Monkey and uses the event-driven programming methodology. Given different exploration strategies, it can be extended to perform different dynamic analyses, such as fraud detection, stress testing, and permission usage profiling.

A³E [47] systematically explores 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.

This study tried to include several other Android testing techniques, such as Mobi-GUITAR [41], JPF-Android [151], and SwiftHand [54]. However, I could not successfully
configure and install them. Problems included missing executable files, missing dependent libraries, and missing configuration files.

### 5.3.6 Experimental Results

This section presents the experimental results, concludes the findings to address the research questions, and discusses key findings.

RQ3: **How effective is Android mutation analysis in testing Android apps?**

**Specifically, how many faults can be detected by mutation-generated tests?**

Table 5.10 summarizes the numbers of naturally occurring faults detected by Android mutation testing. The column *Number of Faults* provides the number of naturally occurring faults for each experimental subject app. The column *Detected by Android Mutation* lists the number of naturally occurring faults detected by mutation-adequate tests. All the faults in JustSit, TippyTipper, and WorldClock were detected. Both AlarmKlock and AndroidomaticKeyer had one fault undetected. Only one in four faults of MunchLife was found. Overall, mutation-adequate tests detected 18 out of 25 naturally occurring faults.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Number of Faults</th>
<th>Detected by Android Mutation</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>7</td>
<td>6</td>
<td>85.71%</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td>3</td>
<td>2</td>
<td>66.67%</td>
</tr>
<tr>
<td>Jamendo</td>
<td>6</td>
<td>4</td>
<td>66.67%</td>
</tr>
<tr>
<td>JustSit</td>
<td>1</td>
<td>1</td>
<td>100.00%</td>
</tr>
<tr>
<td>MunchLife</td>
<td>4</td>
<td>1</td>
<td>25.00%</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>3</td>
<td>3</td>
<td>100.00%</td>
</tr>
<tr>
<td>WorldClock</td>
<td>1</td>
<td>1</td>
<td>100.00%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>25</strong></td>
<td><strong>18</strong></td>
<td><strong>72.00%</strong></td>
</tr>
</tbody>
</table>

Table 5.11 shows the numbers of crowdsourced faults detected by Android mutation testing. The columns are the same as Table 5.11. All 47 crowdsourced faults in TippyTipper
were detected by the mutation-adequate tests, the highest among all the experimental subjects. Two of 59 faults in AlarmKlock were missed, the second highest percentage in fault detection. Only 28 out of 56 faults in Lolcat Builder were found by the mutation-adequate tests, the lowest percentage in fault detection. Overall, the mutation-adequate tests detected 360 out of 437 crowdsourced faults. Note that all these 437 crowdsourced faults are non-mutant faults.

Table 5.11: Numbers and Percentages of Crowdsourced Faults Detected by Android Mutation Testing

<table>
<thead>
<tr>
<th>Apps</th>
<th>Number of Faults</th>
<th>Detected by Android Mutation</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>59</td>
<td>57</td>
<td>96.61%</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td>25</td>
<td>17</td>
<td>68.00%</td>
</tr>
<tr>
<td>Jamendo</td>
<td>40</td>
<td>36</td>
<td>90.00%</td>
</tr>
<tr>
<td>JustSit</td>
<td>59</td>
<td>52</td>
<td>88.14%</td>
</tr>
<tr>
<td>Lolcat Builder</td>
<td>56</td>
<td>28</td>
<td>50.00%</td>
</tr>
<tr>
<td>MunchLife</td>
<td>52</td>
<td>41</td>
<td>78.85%</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>47</td>
<td>47</td>
<td>100.00%</td>
</tr>
<tr>
<td>Tipster</td>
<td>64</td>
<td>51</td>
<td>79.69%</td>
</tr>
<tr>
<td>WorldClock</td>
<td>35</td>
<td>31</td>
<td>88.57%</td>
</tr>
<tr>
<td>TOTAL</td>
<td>437</td>
<td>360</td>
<td>82.38%</td>
</tr>
</tbody>
</table>

RQ4: How effectively do four other Android testing techniques test Android apps? Specifically, how many of the faults can be detected by four other Android testing techniques?

Table 5.12 shows the numbers of naturally occurring faults detected by the four existing Android testing techniques. Each column with a tool’s name presents the number of faults detected by this tool. Dynodroid detected four out of seven naturally occurring faults in AlarmKlock, and Monkey found two for the same subject. Two other Android testing tools, PUMA and A³E, detected only one fault in AlarmKlock. Both Dynodroid and Monkey found one out of three faults in AndroidomaticKeyer, but PUMA and A³E did not find any. These four tools could not find any faults in JustSit, MunchLife, and TippyTipper. Overall,
Dynodroid detected the most naturally occurring faults: 7 out of 25. Monkey detected six, PUMA found three, and A³E discovered only one.

Table 5.13 provides the results of crowdsourced faults detected by the four existing Android testing techniques. Monkey found 31 out of 59 faults in AlarmKlock, and 26 out of 35 faults in WorldClock. Both are the highest among all the four tools. Dynodroid detected the most faults in three other subjects. It found 16 of 25 in AndroidomaticKeyer, 22 of 52 in MunchLife, and 22 of 64 in Tipster. Not surprisingly, as all the four tools are designed and implemented with different strategies, each one has its strengths and weaknesses at detecting faults. Overall, Dynodroid also detected the most crowdsourced faults: 138 out of 437. Monkey found 130, PUMA detected 121, and A³E discovered 79.

**RQ5: Is there any difference between using naturally occurring faults and using crowdsourced faults in empirical evaluations?**

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, it is not possible to conclude that either population of faults led to different results.

### 5.3.7 Statistical Analysis

A one-tailed Wilcoxon signed-rank test [156] with the statistical significance level $\alpha = 0.05$ was used to compare the paired numbers of the naturally occurring faults detected by the four Android testing tools and Android mutation testing. For $Hypotheses_A$, the $W_{value}$ was 0 for both pairs. Because the sample size is less than 10, and at $p \leq 0.05$ (the statistical significance level $\alpha = 0.05$), $W_{value}$ is smaller than the critical value, $W_{critical}$, so the null hypothesis $H_0$ for $Hypotheses_A$ is rejected. That is, the experimental results show that the fault detection effectiveness of Android mutation testing for naturally occurring faults is significantly greater than Monkey, Dynodroid, PUMA, and A³E.

Again, this study used the one-tailed Wilcoxon signed-rank test ($\alpha = 0.05$) to compare the paired numbers of the hand-seeded faults detected by the four Android testing tools and Android mutation testing. For $Hypotheses_B$, the $W_{value}$ was 0 for both pairs. At
<table>
<thead>
<tr>
<th>Apps</th>
<th># Faults</th>
<th>Monkey</th>
<th>% by Monkey</th>
<th>Dynodroid</th>
<th>% by Dynodroid</th>
<th>PUMA</th>
<th>% by PUMA</th>
<th>A³E</th>
<th>% by A³E</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmKlock</td>
<td>7</td>
<td>2</td>
<td>28.57%</td>
<td>4</td>
<td>57.14%</td>
<td>1</td>
<td>14.29%</td>
<td>1</td>
<td>14.29%</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td>3</td>
<td>1</td>
<td>33.33%</td>
<td>1</td>
<td>33.33%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Jamendo</td>
<td>6</td>
<td>2</td>
<td>33.33%</td>
<td>2</td>
<td>33.33%</td>
<td>2</td>
<td>33.33%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>JustSit</td>
<td>1</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>MunchLife</td>
<td>4</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>3</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>WorldClock</td>
<td>1</td>
<td>1</td>
<td>100.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>25</strong></td>
<td><strong>6</strong></td>
<td><strong>24.00%</strong></td>
<td><strong>7</strong></td>
<td><strong>28.00%</strong></td>
<td><strong>3</strong></td>
<td><strong>12.00%</strong></td>
<td><strong>1</strong></td>
<td><strong>4.00%</strong></td>
</tr>
<tr>
<td>Apps</td>
<td># Faults</td>
<td>Monkey</td>
<td>% by Monkey</td>
<td>Dynodroid</td>
<td>% by Dynodroid</td>
<td>PUMA</td>
<td>% by PUMA</td>
<td>A^3E</td>
<td>% by A^3E</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>--------</td>
<td>-------------</td>
<td>-----------</td>
<td>----------------</td>
<td>------</td>
<td>-----------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>AlarmKlock</td>
<td>59</td>
<td>31</td>
<td>52.54%</td>
<td>20</td>
<td>33.90%</td>
<td>26</td>
<td>44.07%</td>
<td>1</td>
<td>1.69%</td>
</tr>
<tr>
<td>AndroidomaticKeyer</td>
<td>25</td>
<td>5</td>
<td>20.00%</td>
<td>16</td>
<td>64.00%</td>
<td>4</td>
<td>16.00%</td>
<td>1</td>
<td>4.00%</td>
</tr>
<tr>
<td>Jamendo</td>
<td>40</td>
<td>7</td>
<td>17.50%</td>
<td>22</td>
<td>55.00%</td>
<td>22</td>
<td>55.00%</td>
<td>22</td>
<td>55.00%</td>
</tr>
<tr>
<td>JustSit</td>
<td>59</td>
<td>16</td>
<td>27.12%</td>
<td>7</td>
<td>11.86%</td>
<td>26</td>
<td>44.07%</td>
<td>15</td>
<td>25.42%</td>
</tr>
<tr>
<td>Lolcat Builder</td>
<td>56</td>
<td>3</td>
<td>5.36%</td>
<td>6</td>
<td>10.71%</td>
<td>19</td>
<td>33.93%</td>
<td>19</td>
<td>33.93%</td>
</tr>
<tr>
<td>MunchLife</td>
<td>52</td>
<td>19</td>
<td>36.54%</td>
<td>22</td>
<td>42.31%</td>
<td>5</td>
<td>9.62%</td>
<td>3</td>
<td>5.77%</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>47</td>
<td>10</td>
<td>21.28%</td>
<td>11</td>
<td>23.40%</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Tipster</td>
<td>64</td>
<td>13</td>
<td>20.31%</td>
<td>22</td>
<td>34.38%</td>
<td>7</td>
<td>10.94%</td>
<td>12</td>
<td>18.75%</td>
</tr>
<tr>
<td>WorldClock</td>
<td>35</td>
<td>26</td>
<td>74.29%</td>
<td>12</td>
<td>34.29%</td>
<td>12</td>
<td>34.29%</td>
<td>6</td>
<td>17.14%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>437</strong></td>
<td><strong>130</strong></td>
<td><strong>29.75%</strong></td>
<td><strong>138</strong></td>
<td><strong>31.58%</strong></td>
<td><strong>121</strong></td>
<td><strong>27.69%</strong></td>
<td><strong>79</strong></td>
<td><strong>18.08%</strong></td>
</tr>
</tbody>
</table>
\( p \leq 0.05 \), \( W_{value} \) is smaller than the critical value, \( W_{critical} \). Thus, the null hypothesis \( H_0 \) for \( Hypotheses_B \) is rejected, too. That is, the experimental results show that the fault detection effectiveness of Android mutation testing for the hand-seeded faults is significantly greater than four other Android testing techniques.

In other words, for both groups of faults, the Android mutation tests found more faults than the four other 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. Figure 5.10 and 5.11 provide two more clear comparisons among all the tools used in this evaluation.

For the first time, Android app developers have a strong technique to design effective tests or evaluate the quality of existing tests.

### 5.3.8 Analysis of Undetected Faults

According to Tables 5.10 and 5.11, around 18% of hand-seeded faults and 28% of naturally occurring faults were not detected by the Android mutation-adequate tests. This section analyzes these undetected faults.

Only 50% of the hand-seeded faults in Lolcat Builder were detected by the Android mutation-adequate tests. Lolcat Builder manipulates images by embedding captions into them, but the tests designed in this study did not check images to decide whether tests failed or not. For example, Figure 5.12 shows two undetected faults of Lolcat Builder. The screenshot on the left is the correct version, in which “Hello world” was added at the top left corner of the image as the caption. The first fault in the middle of Figure 5.12 could not display the correct caption, and the second fault on the right of Figure 5.12 embedded the caption at a wrong location (bottom right of the image). In the experiment, the tests actually caused the apps under test to produce such incorrect images, but the test oracles were insufficient to “see” and verify the images. Also, if the font or color of the embedded caption is mistakenly displayed, no test oracle can notice. That is, the faults in the apps
Figure 5.10: A Comparison of Fault Detection Effectiveness with Naturally Occurring Faults

were propagated, but not observed.

Similarly, only 68% of the hand-seeded faults in AndroidomaticKeyer were detected by the Android mutation-adequate tests. AndroidomaticKeyer produces Morse code for user-entered text and plays it through speakers. Figure 5.13 shows an example undetected fault. No matter what test inputs were provided, no tests oracle could “hear,” capture, and verify the audio of the Morse code played through the speaker of devices.

Overall, this is not a problem with Android mutation testing, 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 [105] discussed this problem in the context of test oracles.

In addition to the test oracle problem, testing randomness is another area that the Android mutation-adequate tests did not do well. Randomness shuffles conditions to make the game process never the same as the previous. Many games rely on randomness to increase entertainment, replayability, and unpredictability. Figure 5.14 shows an example of incorporating randomness in MunchLife, one of the subjects in this experiment. Players of MunchLife need to roll a die to determine the next move in the game. However, no test oracle can verify whether the game displays the die correctly, since (1) the die is technically an image that falls into the previous observability problem, (2) the process of rolling a die
is random, and any face of the die can be a possible and correct result. If there is a fault in the implementation of the random process, such as only displaying one face of the die, the fault cannot be detected. Or, if there is a fault in displaying the die, no tests in this experiment can detect it due to the observability problem.

Some faults can only be revealed when one app broadcasts an Intent to other apps, which the Android mutation-adequate tests in this study did not do. Figure 5.15 shows an example inter-app event in Lolcat Builder. The left screenshot in Figure 5.15 shows that after constructing an image with caption, Lolcat Builder lets users share the image via email or message. Users can click on “Messaging,” and then the Android system should launch the default message client of the device, with a new message created including the image (the right screenshot in Figure 5.15). However, the Android mutation-adequate tests were confined in the current app, and not able to provide any input for email, message, or other apps.
5.3.9 An Additional Common Fault

An additional common fault across several subjects was identified in this study. Many apps include a “settings” or “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. Figure 5.16 shows an example fault. In MunchLife, users can customize a max level, and whoever reaches the max level wins the game. The left screenshot in Figure 5.16 shows that the max level was set to 10. However, in the faulty app (the right screenshot in Figure 5.16), the user achieved a level of 12, which had exceeded the max level. Undoubtedly, the modified setting was not put into effect.
Additional mutation operators should be designed to encourage testers to design tests to ensure the settings menu works correctly.

5.3.10 Threats to Validity

Similar to most experiments in software engineering, this empirical evaluation has several threats to validity, which could potentially influence the experimental results. Even though some of these threats may not be completely avoided, this evaluation took measures to minimize their influence.

Internal validity: One potential threat to internal validity is that only one set of
Android 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 [131].

Also, this study could not guarantee that the crowdsourced faults were representative faults. To avoid any possible bias regarding this threat, this study used a rigorous selection process to review and recruit every participant, as discussed in Section 5.3.4. All the participants were invited to create faults based on their own Android development experience. They were not provided any instructions or guidelines. In addition, the crowdsourced faults that were the same as mutants were eliminated. This experimental study assumed that all
the subject apps do not contain any faults before hand-seeding faults to them.

In this study, all the naturally occurring faults were collected manually, every crowd-sourced fault was hand-inspected, and equivalent mutants were identified by hand, too. All the calculation and statistical analysis in this experimental study were conducted using Microsoft Excel. Manual work could introduce mistakes into the artifacts that may affect the final results.

**External validity:** This study cannot guarantee that the subject apps are representative. To ameliorate this, this study selected the subject apps that were diverse in size, functionality, and features. Also, all of them have been used as subjects by other researchers.

**Construct validity:** The implementation of muDroid and the associated mutation
operators may include software faults. The Android testing tools were obtained from their websites, and configured and installed in the experimental environment of this study. These four tools, and the construction of the experimental environment may include faults. In this study, muDroid and the experimental environment were constantly tested to make sure they work correctly. This study assumed the four Android testing tools worked as expected and did not contain faults.

5.4 Experimental Evaluation of Redundancy in Android Mutation Testing

The two previous experimental evaluations explored the feasibility of Android mutation testing and evaluated its fault detection effectiveness using both naturally occurring faults and crowdsourced faults. The results show that Android mutation testing is effective at detecting both types of software faults.

However, a major hurdle of Android mutation testing is its cost, in terms of several aspects. Even though mobile devices have already evolved from slow devices with tiny, low-resolution, and black-white screens that are only capable of voice calling, texting, and browsing news without any pictures or animations, to portable computers and entertainment terminals, most Android emulators and devices work much slower than personal computers. Therefore, testing Android apps takes more execution time than testing software on desktops and laptops. In addition, testing traditional software is conducted on one single computer or server, while testers for Android apps need to design and compile tests on a computer, then deploy to a connected Android device or emulator. The compiling, deploying, and transmitting also take time. Moreover, while Android mutation testing has been found to be effective at designing high-quality test cases and assessing test cases generated by other testing techniques, the number of mutants that need to be executed increases the cost of Android mutation testing. For example, if an Android app has 1,000 mutants, and the tester creates 20 test cases, and each test takes an average of one minute to run, it will take
13.8 days to finish the entire mutation analysis process.

The motivation of this experimental evaluation was to speed up mutation testing by finding redundant mutation operators that can be excluded from Android mutation testing, while still maintaining the effectiveness of Android mutation testing. Specifically, this experimental study analyzed redundancy among the 19 Java traditional mutation operators and the 17 Android mutation operators. Some of these mutation operators may be redundant and do not contribute to the quality of tests. If redundant mutation operators can be identified and excluded, the cost of mutation will be reduced. All mutants generated from the same mutation operator are of the same type. So, this study determines whether mutants of one type are killed by the tests designed to kill mutants of other types.

In particular, the following research questions were addressed:

**RQ6:** How many mutants of one particular type can be killed by tests created to kill another type of mutants?

**RQ7:** Which types of mutants are less likely killed by tests created to kill other types of mutants?

**RQ8:** Can any mutation operator be excluded or improved without significantly reducing effectiveness?

### 5.4.1 Experimental Subjects

This experimental evaluation reused four open source Android apps from previous experiments, including 12 Android classes and their XML layout and configuration files: *JustSit* [22], *MunchLife* [26], *TippyTipper* [33], and *Tipster* [59]. These subject apps were described in Section 5.2.1. Table 5.14 provides an overview of the files used in this experimental evaluation. The Lines of Code (LOC) in the Java files were measured with Metrics [25], and the numbers of XML nodes (elements) in the XML layout and manifest (configuration) files were counted with a specific XML parser plugin of muDroid.

The 19 Java traditional method-level mutation operators from muJava [114] generated 1,947 muJava mutants, and the 17 Android mutation operators developed in this research
<table>
<thead>
<tr>
<th>Apps</th>
<th>Components</th>
<th>LOC</th>
<th>XML Elements</th>
<th>muJava Mutants</th>
<th>Android Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>JustSit</td>
<td>JustSit.java</td>
<td>444</td>
<td>394</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>About.java</td>
<td>48</td>
<td>9</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>about.xml</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RunTimer.java</td>
<td>99</td>
<td>131</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>run_timer.xml</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JsSettings.java</td>
<td>61</td>
<td>28</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>settings.xml</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>14</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MunchLife</td>
<td>MunchLifeActivity.java</td>
<td>384</td>
<td>534</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Settings.java</td>
<td>68</td>
<td>47</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>preferences.xml</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AndroidManifest.xml</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TippyTipper</td>
<td>TippyTipper.java</td>
<td>239</td>
<td>105</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SplitBill.java</td>
<td>134</td>
<td>124</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SplitBill.xml</td>
<td></td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total.java</td>
<td>279</td>
<td>231</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total.xml</td>
<td></td>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>About.java</td>
<td>30</td>
<td>4</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>About.xml</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Settings.java</td>
<td>61</td>
<td>13</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Tipster</td>
<td>TipsterActivity.java</td>
<td>297</td>
<td>327</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td></td>
<td>main.xml</td>
<td></td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>2144</td>
<td>204</td>
<td>1947</td>
<td>1018</td>
</tr>
</tbody>
</table>
generated 1,018 Android mutants. The number of muJava mutants ranged from four for About.java in TippyTipper to 534 for MunchLifeActivity.java in MunchLife, and the number of Android mutants ranges from one for AndroidManifest.xml in MunchLife to 258 for JustSit.java in JustSit.

5.4.2 Redundancy Scores

The mutation-adequate test set $T_i$ includes tests that are specifically designed to kill all the mutation in type $i$. To quantify the redundancy among Java traditional method-level mutation operators and Android mutation operators, Praphamontripong and Offutt [140] defined the redundancy score $r_{i,j}$ to be:

\[
\text{Redundancy Score: } r_{i,j} = \frac{m_{i,j}}{M_j} \times 100\%
\]  

(5.1)

where:

- $m_{i,j} =$ Number of mutants of type $j$ killed by the mutation-adequate test set $T_i$
- $M_j =$ Total number of non-equivalent mutants of type $j$

In other words, the redundancy score $r_{i,j}$ is the percentage of the mutants of type $j$ killed by test set that is adequate for type $i$. For example, a program has 100 non-equivalent Relational Operator Replacement (ROR) mutants and 200 non-equivalent Shortcut Arithmetic Operator Insertion (AOIS) mutants. A tester designs a test set that kills all the non-equivalent AOIS mutants, getting an AOIS mutation-adequate test set. If this AOIS mutation-adequate test set also kills 60 ROR mutants, the redundancy score $r_{AOIS,ROR}$ in this program is $60 \div 100 = 60\%$.

Note that in one subject app, according to the definition above, every possible pair of mutation operators has a redundancy score. Then, across all the subject apps in an experimental evaluation, there are multiple redundancy scores for the same pair of mutation operators with different values. For example, $r_{AOIS,ROR}$ may be 60% in subject $s_1$, 50% in
s_2, and 40% in s_3. Consequently, a score that can represent the overall redundancy relationship is required. Praphamontripong and Offutt [140] defined two types of redundancy scores: overall redundancy score and average redundancy score. The overall redundancy score \( r_{\text{overall},i,j} \) of a mutation operator is the percentage of the cumulative number of killed mutants of type \( j \) from all the subjects killed by an adequate test set that is specifically designed to kill the mutants of type \( i \), while the average redundancy score \( r_{\text{average},i,j} \) of a mutation operator is an average value of \( r_{i,j} \) in each subject.

Redundancy score is a quantitative indicator of whether a mutation operator is redundant or not. For example, if a mutation operator has a redundancy score of 0%, it means no tests that were designed to kill other types of mutants could kill the mutants of this type. That is, this mutation operator is not redundant at all. However, if a mutation operator has a redundancy score of 100% for the tests that are specifically designed to kill mutants of another type, it means this operator is totally redundant and does not contribute to the quality of tests. Excluding it from the mutation analysis can reduce cost without reducing effectiveness. If a mutation operator has a redundancy score of 50%, half of the mutants generated by this operator are killed by the tests designed for other types of mutants.

Sometimes, a subject app may not generate every type of mutants as it does not have the necessary features required by certain mutation operators. Then, no tests will be designed for this mutation type.

### 5.4.3 Experimental Procedure

This study includes four steps to obtain the redundancy scores among all the Android mutation operators. Figure 5.17 shows a general experimental procedure:

1. **Generate mutants:** Given a subject, the 19 Java traditional method-level mutation operators and the 17 Android mutation operators were used to generate mutants, denoted by \( m_n \), that is, mutants created by the operator \( n \).

2. **Eliminate equivalent mutants and design tests:** For each set of mutants \( m_n \),
all equivalent mutants were identified and eliminated by hand. Then, a set of test cases was designed to kill all the non-equivalent mutants, denoted by \( t_n \), that is, tests designed to kill the mutants of type \( n \).

3. **Execute tests:** For each set of test cases \( t_n \), execute it on every type of mutants, from \( m_1 \) to \( m_n \).

4. **Compute the redundancy scores:** For each pair of mutation operators and for each subject app, the redundancy score \( r_{i,j} \) was computed, that is, the percentage of the mutants of type \( j \) killed by the test cases \( t_i \) specifically designed to kill the mutants of type \( i \). Then, to get an overview across all the subjects in the experiment, an overall redundancy score and an average redundancy score were computed for each type of mutant.
5.4.4 Experiment Results and Discussion

This section presents the experimental results and discusses key findings.

RQ6: How many mutants of one particular type can be killed by tests created to kill another type of mutants?

Table 5.15 shows the average redundancy scores across all the subject apps obtained from the experiment. If across all the subject apps, the mutants of type $i$ and the mutants of type $j$ were never generated from the same subject at the same time, the Android mutation-adequate test sets of these two types will not have the chance to execute against the mutants of the other. These are indicated with “n/a.”

Four Java traditional method-level mutation operators, ASRS, LOD, SOR, and AODS, did not generate any mutants, and four Android mutation operators, ETR, LCM, SMDL, and WCD, did not generate any mutants. Thus, these mutation operators are not listed in Table 5.15.

The WakeLock Release Deletion (WRD) mutation operator deletes every method call to release() to force the Android app not to release the wake lock. It mimics a typical energy bug, which commonly happens when the app does not release system resources during an idle state. To kill WRD mutants, testers need to use the `dumpsys` command in command line, which can capture system information from Android devices. This command can help testers identify active wake locks after the app under test has been closed. Figure 4.21 in Section 4.2.6 showed an example of using `dumpsys` to kill WRD mutants. Since using the `dumpsys` command is a unique way of designing tests and cannot kill other types of mutants, WRD mutants and tests were not used in this experiment.

The Activity Permission Deletion (APD) mutation operator deletes permission requests from configuration files one at a time, because some apps aggressively request unnecessary permissions, which may create security vulnerabilities in Android systems. If an APD mutant cannot be killed by any tests, it means that the app asked for a permission it did not need. Therefore, testers need to test whether a permission request is necessary to the app. Instead of designing tests to kill mutants, the principle of APD is to try all possible
<table>
<thead>
<tr>
<th>Table 5.15: Average Redundancy Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AOIS</strong></td>
</tr>
<tr>
<td>0.108</td>
</tr>
</tbody>
</table>

**malava Mutation Operator**

| **AOIS** | **AOID** | **AOIU** | **AOIR** | **COD** | **COR** | **COL** | **CRI** | **CRO** | **DOL** | **DOM** | **DOR** | **EOL** | **EOR** | **FOL** | **FOR** | **GOL** | **GOR** | **HOL** | **HOR** | **IOL** | **IOR** | **JOL** | **JOR** | **KOL** | **KOR** | **LOR** | **LOR** | **MLR** | **MRD** | **MRD** | **NOR** | **NOR** | **OLR** | **OLR** | **ORL** | **ORL** | **TOL** | **TOL** | **TDR** | **TDR** | **TEN** | **TEN** | **VOL** | **VOL** | **VOR** | **VOR** | **WOL** | **WOL** | **WOR** | **WOR** |
| 1.000 | 0.839 | 0.573 | 0.730 | 1.400 | 0.537 | 0.524 | 0.630 | 0.000 | 0.481 | 0.504 | 0.492 | 0.599 | 0.392 | 0.590 | 0.500 | 0.612 | 0.331 | 0.330 | 0.000 | 0.012 | 0.004 | 0.367 | 0.759 | 0.000 | 0.000 | 0.000 |

| **Android Mutation Operator**

| **AOIS** | **AOID** | **AOIU** | **AOIR** | **COD** | **COR** | **COL** | **CRI** | **CRO** | **DOL** | **DOM** | **DOR** | **EOL** | **EOR** | **FOL** | **FOR** | **GOL** | **GOR** | **HOL** | **HOR** | **IOL** | **IOR** | **JOL** | **JOR** | **KOL** | **KOR** | **LOR** | **LOR** | **MLR** | **MRD** | **MRD** | **NOR** | **NOR** | **OLR** | **OLR** | **ORL** | **ORL** | **TOL** | **TOL** | **TDR** | **TDR** | **TEN** | **TEN** | **VOL** | **VOL** | **VOR** | **VOR** | **WOL** | **WOL** | **WOR** | **WOR** |
| 1.000 | 0.839 | 0.573 | 0.730 | 1.400 | 0.537 | 0.524 | 0.630 | 0.000 | 0.481 | 0.504 | 0.492 | 0.599 | 0.392 | 0.590 | 0.500 | 0.612 | 0.331 | 0.330 | 0.000 | 0.012 | 0.004 | 0.367 | 0.759 | 0.000 | 0.000 | 0.000 |

**Average**

| **AOIS** | **AOID** | **AOIU** | **AOIR** | **COD** | **COR** | **COL** | **CRI** | **CRO** | **DOL** | **DOM** | **DOR** | **EOL** | **EOR** | **FOL** | **FOR** | **GOL** | **GOR** | **HOL** | **HOR** | **IOL** | **IOR** | **JOL** | **JOR** | **KOL** | **KOR** | **LOR** | **LOR** | **MLR** | **MRD** | **MRD** | **NOR** | **NOR** | **OLR** | **OLR** | **ORL** | **ORL** | **TOL** | **TOL** | **TDR** | **TDR** | **TEN** | **TEN** | **VOL** | **VOL** | **VOR** | **VOR** | **WOL** | **WOL** | **WOR** | **WOR** |
| 0.569 | 0.450 | 0.465 | 0.538 | 0.387 | 0.468 | 0.514 | 0.125 | 0.319 | 0.361 | 0.404 | 0.382 | 0.576 | 0.177 | 0.652 | 0.064 | 0.561 | 0.530 | 0.626 | 0.743 | 0.125 | 0.907 | 0.375 |
tests to identify those APD mutants that can never be killed by any tests. Thus, APD was not included in this experiment either.

**RQ7: Which types of mutants are less likely to be killed by tests created to kill other types of mutants?**

Some Android mutants are very hard to kill, and they are sometimes grouped by type. Table 5.16 extracts the average redundancy scores of Fail on Back (FOB) mutants. On average, only 6.4% of Fail on Back (FOB) mutants were killed by the mutation adequate test sets of other mutation operators, with the highest redundancy score of 33.3%. FOB injects a “Fail on Back” event handler into every Activity class. Since Android apps are event-based programs, their execution flows rely heavily on events initiated by different user actions. The Back button lets users move backward to the previous Activity, interrupting the usual execution flow. It is usually not on the happy path from the perspective of software design, and results in a common fault of Android apps, that is the crash when the Back button is clicked. To kill FOB mutants, testers need to design tests that press the Back button at least once at every Activity. However, in this experiment, very few tests designed for other mutation operators included the user action of clicking the Back button.

The TextView Deletion (TVD) mutation operator is another type for which very few mutants were killed in the experiment. Table 5.17 extracts the average redundancy scores of TVD. On average, less than 1% of TVD mutants were killed by the mutation adequate test sets of other mutation operators, with the highest redundancy score of 8.3%. Figure 4.13 in Section 4.2.3 provided an example screenshot of a TVD mutant. Since TextView widgets cannot be edited by users, they usually do not associate with any user events, nor require any event handlers from the implementation of the app. However, TextView widgets are widely used by developers to present essential information. TVD deletes TextView widgets from screens one at a time. Killing a TVD mutant needs a test to ensure that TextView widget displays correct information. Very few tests checked TextView widgets' contents, unless the TextView widget was used to display some variable results, such as a tip amount.

The Orientation Lock (ORL) mutation operator had very few mutants killed in the
Table 5.16: Average Redundancy Scores of Fail on Back (FOB)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Fail on Back (FOB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.000</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.167</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>0.125</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.000</td>
</tr>
<tr>
<td>test_CDL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_COD</td>
<td>0.000</td>
</tr>
<tr>
<td>test_COI</td>
<td>0.200</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.333</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.125</td>
</tr>
<tr>
<td>test_LOR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ODL</td>
<td>0.143</td>
</tr>
<tr>
<td>test_ROR</td>
<td>0.200</td>
</tr>
<tr>
<td>test(SDL</td>
<td>0.250</td>
</tr>
<tr>
<td>test(SDL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_BWD</td>
<td>0.000</td>
</tr>
<tr>
<td>test_BWS</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ECR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_FOB</td>
<td>———</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.000</td>
</tr>
<tr>
<td>test_IPR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ITR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_MDL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.000</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.064</strong></td>
</tr>
</tbody>
</table>
Table 5.17: Average Redundancy Scores of TextView Deletion (TVD)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>TextView Deletion (TVD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.000</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.000</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>0.000</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.000</td>
</tr>
<tr>
<td>test_CD</td>
<td>0.050</td>
</tr>
<tr>
<td>test_COD</td>
<td>0.000</td>
</tr>
<tr>
<td>test_COI</td>
<td>0.000</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.000</td>
</tr>
<tr>
<td>test_LOR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ODL</td>
<td>0.042</td>
</tr>
<tr>
<td>test_ROR</td>
<td>0.000</td>
</tr>
<tr>
<td>test(SDL)</td>
<td><strong>0.083</strong></td>
</tr>
<tr>
<td>test_VDL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_BWD</td>
<td>0.000</td>
</tr>
<tr>
<td>test_BWS</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ECR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.000</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.000</td>
</tr>
<tr>
<td>test_IPR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ITR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_MDL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_TVD</td>
<td>——</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.007</strong></td>
</tr>
</tbody>
</table>
experiment. Table 5.18 lists the average redundancy scores of ORL. On average, only 2.5% of Orientation Lock (ORL) mutants were killed by the mutation adequate test sets of other mutation operators, with the highest redundancy score of 12.5%. Most mobile devices have the unique feature of being able to change the screen orientation. To use this feature, many apps change their layout of the GUI when the orientation changes. For example, Figure 1.9 in Section 1.2 provided an example of a simple calculator (left) with portrait orientation that becomes a scientific calculator (right) when switched to landscape orientation. However, different screen sizes and resolutions, and different devices, make switching the orientation difficult for the developers and lead to many faults in Android apps. ORL mutants freeze the orientation of an Activity by inserting a special locking statement into the source code, so that no switching actions can be accepted by the app. To kill ORL mutants, testers need to design tests that explicitly change the orientation, then check whether the GUI structure is displayed as expected after switching the orientation. Again, in this experiment, no other mutation operators consider switching the screen orientation.

RQ8: Are any Android mutation operators redundant enough to be excluded, or can any be improved? In particular, can the mutants generated from this mutation operator always be killed by tests created to kill another type of mutant?

According to the results, several mutation operators generated mutants that were easily killed by the tests designed to kill other types of mutants.

Among the 17 Android mutation operators, the Activity Lifecycle Method Deletion (MDL) mutation operator has the highest mean redundancy score (74.3%). Table 5.19 lists the average redundancy scores of MDL. Android operating systems require all components in Android apps to behave according to a pre-defined lifecycle. Figure 1.5 in Section 1.2 provided an overview of the Activity component lifecycle. If developers want to define a specific behavior when an Activity switches its state, they must follow the lifecycle and override correct methods in it. For example, after an Activity is launched, three methods, `onCreate()`, `onStart()`, and `onResume()`, need to be executed sequentially before the user
Table 5.18: Average Redundancy Scores of Orientation Lock (ORL)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Orientation Lock (ORL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.000</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.021</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>0.016</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.025</td>
</tr>
<tr>
<td>test_CDL</td>
<td>0.025</td>
</tr>
<tr>
<td>test_COD</td>
<td>0.000</td>
</tr>
<tr>
<td>test_COI</td>
<td>0.025</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.042</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.016</td>
</tr>
<tr>
<td>test_LOR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ODL</td>
<td>0.018</td>
</tr>
<tr>
<td>test_ROR</td>
<td>0.125</td>
</tr>
<tr>
<td>test(SDL)</td>
<td>0.078</td>
</tr>
<tr>
<td>test_VDL</td>
<td>0.025</td>
</tr>
<tr>
<td>test_BWD</td>
<td>0.031</td>
</tr>
<tr>
<td>test_BWS</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ECR</td>
<td>0.042</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.000</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.018</td>
</tr>
<tr>
<td>test_JPR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ITR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_MDL</td>
<td>0.016</td>
</tr>
<tr>
<td>test_ORL</td>
<td>——</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.083</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.025</strong></td>
</tr>
</tbody>
</table>
Table 5.19: Average Redundancy Scores of Activity Lifecycle Method Deletion (MDL)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Activity Lifecycle Method Deletion (MDL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.750</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.875</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>0.781</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.850</td>
</tr>
<tr>
<td>test_CDL</td>
<td><strong>0.900</strong></td>
</tr>
<tr>
<td>test_COD</td>
<td><strong>1.000</strong></td>
</tr>
<tr>
<td>test_COI</td>
<td>0.850</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.833</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.781</td>
</tr>
<tr>
<td>test_LOR</td>
<td>0.500</td>
</tr>
<tr>
<td>test_ODL</td>
<td>0.893</td>
</tr>
<tr>
<td>test_ROR</td>
<td>0.850</td>
</tr>
<tr>
<td>test(SDL)</td>
<td><strong>0.906</strong></td>
</tr>
<tr>
<td>test_VDL</td>
<td>0.850</td>
</tr>
<tr>
<td>test_BWD</td>
<td>0.688</td>
</tr>
<tr>
<td>test_BWS</td>
<td><strong>0.917</strong></td>
</tr>
<tr>
<td>test_ECR</td>
<td>0.583</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.281</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.607</td>
</tr>
<tr>
<td>test_IPR</td>
<td>0.500</td>
</tr>
<tr>
<td>test_ITR</td>
<td>0.750</td>
</tr>
<tr>
<td>test_MDL</td>
<td>——</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.219</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.833</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.833</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.743</strong></td>
</tr>
</tbody>
</table>
can see the Activity on the screen. 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. However, many developers use `onCreate()` to define and initialize GUI structures of their apps. After MDL deletes the content of `onCreate()`, no GUI widgets can be displayed for the current Activity. Figure 5.18 shows an example of this situation. The MDL mutant on the right is trivial, and any test case that looks for a GUI widget or initiates a user event can kill this MDL mutant.

A recommendation is that instead of simply deleting the content of `onCreate()`, an alternative implementation is to move the content of `onCreate()` to `onStart()` and `onResume()`. Figure 5.19 gives an example of the recommended implementation. All the code that defines GUI widgets and initializes event handlers has been migrated from `onCreate()` to `onStart()`.
In this way, MDL mutants are no longer trivial. In addition, the only way to kill this new version of MDL mutants is to make the Activity switch among different states, so that different lifecycle methods can be called. Therefore, modified MDL would require testers to design tests to make the Activity switch among different states.

Among the 19 muJava mutation operators, the Unary Arithmetic Operator Deletion (AODU) mutation operator has the highest mean redundancy score (77.1%). Table 5.20 lists the average redundancy scores of AODU. 16 sets of mutation adequate test sets designed to kill other types of mutants killed all AODU mutants, indicated by “1.000” values in Table 5.20. AODU deletes basic unary arithmetic operators in an expression. Figure 5.21 shows an example AODU mutant, in which the minus symbol is deleted. The results indicate that AODU is redundant and can be excluded.

As shown in Table 5.22, the Button Widget Deletion (BWD) has six “1.000” values, which is the second highest among all muJava and Android mutation operators. In fact, all the BWD mutants were killed by the BWS tests. Table 5.22 shows the average redundancy scores of BWD and BWS. Button widgets are used by nearly all Android apps in many ways. BWD deletes buttons one at a time from the XML layout file of the UI. BWS switches the locations of two buttons on the same screen. In this way, the function of a button is unaffected, but the GUI layout looks different from the original version. BWS requires the tester to design tests that deliberately check the location (either relative or absolute) of a button widget. Figure 5.20 gives example BWS and BWD mutants. The BWS mutant (middle) switches the locations of button “7” and “OK.” The BWD mutant (right) deletes button “OK.”

Not surprisingly, when BWS mutants ensure every button is displayed at an expected location, it is also necessarily guaranteed that this button is shown on the screen. Subsumption is used to theoretically compare test criteria: a criterion C1 subsumes another criterion C2, if every test that satisfies C1 is guaranteed to satisfy C2 [43]. Particularly in mutation testing, a mutation operator MO1 subsumes another mutation operator MO2 if a test set that kills all mutants of MO1 is guaranteed to kill MO2. Thus, BWS subsumes
Table 5.20: Average Redundancy Scores of Unary Arithmetic Operator Deletion (AODU)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Unary Arithmetic Operator Deletion (AODU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td></td>
</tr>
<tr>
<td>test_AOIS</td>
<td>1.000</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>1.000</td>
</tr>
<tr>
<td>test_AORB</td>
<td>1.000</td>
</tr>
<tr>
<td>test_CDL</td>
<td>1.000</td>
</tr>
<tr>
<td>test_COD</td>
<td>1.000</td>
</tr>
<tr>
<td>test_CODU</td>
<td></td>
</tr>
<tr>
<td>test_COI</td>
<td>1.000</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.000</td>
</tr>
<tr>
<td>test_LOI</td>
<td>1.000</td>
</tr>
<tr>
<td>test_LOR</td>
<td>1.000</td>
</tr>
<tr>
<td>test_ODL</td>
<td>1.000</td>
</tr>
<tr>
<td>test_ROR</td>
<td>1.000</td>
</tr>
<tr>
<td>test_SDL</td>
<td>1.000</td>
</tr>
<tr>
<td>test_VDL</td>
<td>1.000</td>
</tr>
<tr>
<td>test_BWD</td>
<td>1.000</td>
</tr>
<tr>
<td>test_BWS</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ECR</td>
<td>1.000</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.500</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.500</td>
</tr>
<tr>
<td>test_IPR</td>
<td>1.000</td>
</tr>
<tr>
<td>test_ITR</td>
<td>1.000</td>
</tr>
<tr>
<td>test_MDNL</td>
<td>0.500</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.000</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.500</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.500</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.771</strong></td>
</tr>
</tbody>
</table>
BWD, that is, every test set designed to kill all the BWS mutants can kill all the BWD mutants. As a result, when users include BWS mutants in the Android mutation analysis, excluding BWD mutants will not affect test effectiveness. Note that if an Activity only has
Table 5.21: An Example AODU Mutant

| Original: int x = - y ; | AODU Mutant: int x = y ; |

one button widget, BWS cannot generate any mutants. This is because to achieve switching, the Activity must display at least two buttons. Thus, I recommend disabling BWD when there are BSW mutants, and enabling it otherwise.

The Conditional Operator Deletion (COD) mutation operator also has six “1.000” values (second highest), and the Constant Deletion (CDL) mutation operator has five “1.000” values (third highest). Also, the ODL test sets killed all the mutants of CDL, COD, and the Variable Deletion mutation operator (VDL). Table 5.23 lists the average redundancy scores of CDL, COD, ODL, and VDL. The Operator Deletion mutation operator (ODL) was originally designed by Delamaro et al. [63]. It deletes each arithmetic, relational, logical, bitwise, and shift operator from all expressions. CDL deletes each constant in an expression,
Table 5.22: Average Redundancy Scores of Button Widget Deletion (BWD) and Button Widget Switch (BWS)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Button Widget Deletion (BWD)</th>
<th>Button Widget Switch (BWS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.125</td>
<td>0.333</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.688</td>
<td>0.111</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>0.813</td>
<td>0.167</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.688</td>
<td>0.111</td>
</tr>
<tr>
<td>test_CDL</td>
<td>0.583</td>
<td>0.111</td>
</tr>
<tr>
<td>test_COD</td>
<td>0.250</td>
<td>0.333</td>
</tr>
<tr>
<td>test_COI</td>
<td>0.833</td>
<td>0.417</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.375</td>
<td>0.333</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.813</td>
<td>0.278</td>
</tr>
<tr>
<td>test_LOR</td>
<td>1.000</td>
<td>n/a</td>
</tr>
<tr>
<td>test_ODL</td>
<td>0.792</td>
<td>0.278</td>
</tr>
<tr>
<td>test_ROR</td>
<td>1.000</td>
<td>0.417</td>
</tr>
<tr>
<td>test(SDL)</td>
<td>1.000</td>
<td>0.278</td>
</tr>
<tr>
<td>test_VDL</td>
<td>0.688</td>
<td>0.111</td>
</tr>
<tr>
<td>test_BWD</td>
<td>n/a</td>
<td>0.278</td>
</tr>
<tr>
<td>test_BWS</td>
<td>1.000</td>
<td>n/a</td>
</tr>
<tr>
<td>test_ECR</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.500</td>
<td>0.000</td>
</tr>
<tr>
<td>test_LPR</td>
<td>1.000</td>
<td>n/a</td>
</tr>
<tr>
<td>test_LTR</td>
<td>0.389</td>
<td>0.000</td>
</tr>
<tr>
<td>test_MDL</td>
<td>0.125</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.167</td>
<td>0.333</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.576</strong></td>
<td><strong>0.177</strong></td>
</tr>
</tbody>
</table>
Table 5.23: Average Redundancy Scores of Constant Deletion (CDL), Conditional Operator Deletion (COD), Operator Deletion (ODL), and Variable Deletion (VDL)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Constant Deletion (CDL)</th>
<th>Conditional Operator Deletion (COD)</th>
<th>Operator Deletion (ODL)</th>
<th>Variable Deletion (VDL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.000</td>
<td>0.000</td>
<td>0.053</td>
<td>0.000</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.750</td>
<td>1.000</td>
<td>0.491</td>
<td>0.937</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>0.600</td>
<td>0.750</td>
<td>0.402</td>
<td>0.916</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.750</td>
<td>0.750</td>
<td>0.527</td>
<td>0.947</td>
</tr>
<tr>
<td>test_CDL</td>
<td>——</td>
<td>0.750</td>
<td>0.567</td>
<td>0.702</td>
</tr>
<tr>
<td>test_COD</td>
<td>1.000</td>
<td>——</td>
<td>0.474</td>
<td>0.333</td>
</tr>
<tr>
<td>test_COI</td>
<td>0.500</td>
<td>1.000</td>
<td>0.439</td>
<td>0.579</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.125</td>
<td>0.750</td>
<td>0.277</td>
<td>0.026</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.600</td>
<td>1.000</td>
<td>0.425</td>
<td>0.916</td>
</tr>
<tr>
<td>test_LOR</td>
<td>n/a</td>
<td>n/a</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ODL</td>
<td>1.000</td>
<td>1.000</td>
<td>——</td>
<td>1.000</td>
</tr>
<tr>
<td>test_ROR</td>
<td>1.000</td>
<td>1.000</td>
<td>0.664</td>
<td>0.658</td>
</tr>
<tr>
<td>test(SDL)</td>
<td>1.000</td>
<td>1.000</td>
<td>0.916</td>
<td>0.937</td>
</tr>
<tr>
<td>test_VDL</td>
<td>1.000</td>
<td>0.750</td>
<td>0.527</td>
<td>——</td>
</tr>
<tr>
<td>test_BWD</td>
<td>0.333</td>
<td>0.750</td>
<td>0.316</td>
<td>0.355</td>
</tr>
<tr>
<td>test_BWS</td>
<td>0.000</td>
<td>0.000</td>
<td>0.013</td>
<td>0.018</td>
</tr>
<tr>
<td>test_ECR</td>
<td>0.750</td>
<td>n/a</td>
<td>0.400</td>
<td>0.474</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.000</td>
<td>0.000</td>
<td>0.006</td>
<td>0.011</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.125</td>
<td>0.000</td>
<td>0.125</td>
<td>0.079</td>
</tr>
<tr>
<td>test_IPR</td>
<td>n/a</td>
<td>n/a</td>
<td>0.389</td>
<td>0.000</td>
</tr>
<tr>
<td>test_ITR</td>
<td>0.250</td>
<td>n/a</td>
<td>0.240</td>
<td>0.035</td>
</tr>
<tr>
<td>test_MDL</td>
<td>0.250</td>
<td>0.000</td>
<td>0.023</td>
<td>0.021</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.000</td>
<td>0.000</td>
<td>0.006</td>
<td>0.011</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.200</td>
<td>0.000</td>
<td>0.223</td>
<td>0.211</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.000</td>
<td>0.250</td>
<td>0.147</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.465</strong></td>
<td><strong>0.538</strong></td>
<td><strong>0.319</strong></td>
<td><strong>0.382</strong></td>
</tr>
</tbody>
</table>
and VDL deletes each variable in an expression. Figures 5.24 and 5.25 show example ODL, CDL, and VDL mutants. According to the definitions, it is guaranteed that ODL subsumes CDL and VDL. COD deletes unary conditional operators. Figure 5.26 shows that ODL and COD generate the same mutants. Theoretically, ODL also subsumes COD. Not surprisingly, test cases designed to kill ODL mutants can also kill CDL, COD, and VDL mutants, which means when using ODL, we can exclude CDL, COD, and VDL.

<table>
<thead>
<tr>
<th>Table 5.24: Example ODL Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original: int x = y + 2 ;</td>
</tr>
<tr>
<td>ODL Mutant_2: int x = 2 ;</td>
</tr>
<tr>
<td>Original: int x = - y ;</td>
</tr>
<tr>
<td>Original: if (! isError) {</td>
</tr>
<tr>
<td>x = y ;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.25: An Example CDL and VDL Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original: int x = y + 2 ;</td>
</tr>
<tr>
<td>VDL Mutant: int x = 2 ;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.26: An Example COD Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original: if (! isError) {</td>
</tr>
<tr>
<td>x = y ;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

The Unary Arithmetic Operator Insertion (AOIU) operator inserts a minus sign in front of integer variables. The Logical Operator Insertion (LOI) inserts a bitwise complement operator in front of integer variables. Table 5.27 lists the average redundancy scores of AOIU and LOI. 50.3% of AOIU mutants and 44.9% of LOI mutants were killed by tests.
Table 5.27: Average Redundancy Scores of Unary Arithmetic Operator Insertion (AOIU) and Logical Operator Insertion (LOI)

<table>
<thead>
<tr>
<th>Mutation Adequate Test Set</th>
<th>Unary Arithmetic Operator Insertion (AOIU)</th>
<th>Logical Operator Insertion (LOI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_AODU</td>
<td>0.351</td>
<td>0.286</td>
</tr>
<tr>
<td>test_AOIS</td>
<td>0.819</td>
<td>0.631</td>
</tr>
<tr>
<td>test_AOIU</td>
<td>——</td>
<td>0.681</td>
</tr>
<tr>
<td>test_AORB</td>
<td>0.688</td>
<td>0.561</td>
</tr>
<tr>
<td>test_CDL</td>
<td>0.768</td>
<td>0.645</td>
</tr>
<tr>
<td>test_COD</td>
<td>0.632</td>
<td>0.500</td>
</tr>
<tr>
<td>test_COI</td>
<td>0.471</td>
<td>0.574</td>
</tr>
<tr>
<td>test_COR</td>
<td>0.385</td>
<td>0.505</td>
</tr>
<tr>
<td>test_LOI</td>
<td>0.868</td>
<td>——</td>
</tr>
<tr>
<td>test_LOR</td>
<td>0.176</td>
<td>0.138</td>
</tr>
<tr>
<td>test_ODL</td>
<td>0.847</td>
<td>0.796</td>
</tr>
<tr>
<td>test_ROR</td>
<td>0.543</td>
<td>0.621</td>
</tr>
<tr>
<td>test(SDL)</td>
<td>0.921</td>
<td>0.853</td>
</tr>
<tr>
<td>test_VDL</td>
<td>0.743</td>
<td>0.653</td>
</tr>
<tr>
<td>test_BWD</td>
<td>0.566</td>
<td>0.583</td>
</tr>
<tr>
<td>test_BWS</td>
<td>0.445</td>
<td>0.418</td>
</tr>
<tr>
<td>test_ECR</td>
<td>0.532</td>
<td>0.521</td>
</tr>
<tr>
<td>test_FOB</td>
<td>0.503</td>
<td>0.449</td>
</tr>
<tr>
<td>test_FON</td>
<td>0.578</td>
<td>0.537</td>
</tr>
<tr>
<td>test_IPR</td>
<td>0.353</td>
<td>0.138</td>
</tr>
<tr>
<td>test_ITR</td>
<td>0.471</td>
<td>0.415</td>
</tr>
<tr>
<td>test_MDL</td>
<td>0.520</td>
<td>0.472</td>
</tr>
<tr>
<td>test_ORL</td>
<td>0.503</td>
<td>0.449</td>
</tr>
<tr>
<td>test_TVD</td>
<td>0.449</td>
<td>0.426</td>
</tr>
<tr>
<td>test_TWD</td>
<td>0.518</td>
<td>0.476</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.569</strong></td>
<td><strong>0.514</strong></td>
</tr>
</tbody>
</table>
(test.FOB) designed to kill Fail on Back (FOB) mutants, which are very simple tests that only launch an Activity and click on the Back button.

```java
int level = 1;
current_level.setText ( Integer.toString ( level ) );
```

Original

```java
int level = 1;
current_level.setText ( Integer.toString ( -level ) );
```

AOIU

```java
int level = 1;
current_level.setText ( Integer.toString ( ~level ) );
```

LOI

Figure 5.21: AOIU and LOI Examples

Figure 5.21 gives example AOIU and LOI mutants. In Android apps, each GUI widget is assigned a resource ID that is recorded as an integer number. These resource IDs are stored and managed in XML files. Both AOIU and LOI generate many mutants by mutating the resource IDs in Android apps. Figure 5.22 shows an example where AOIU changes the resource ID of `up_button`. However, once a resource ID is changed and not mapped to its original GUI widget, the Android app will immediately crash after launched, making the mutant trivial and redundant. That is, any test case that launches the app can kill this mutant. Similarly, LOI also generates trivial mutants. Therefore, when using mutation testing for Android apps, I recommend to exclude AOIU and LOI.

```
Button up_button = (Button) findViewById ( R.id.up_button );
```

Original

```
Button up_button = (Button) findViewById ( -R.id.up_button );
```

AOIU

In summary, according to the results of this experiment, I recommend:
1. Exclude AODU, because its highest average redundancy scores

2. Improve the design of MDL, because MDL generates trivial mutants

3. Exclude BWD when using BWS, because BWS subsumes BWD

4. Exclude AOIU and LOI, because around 50% of AOIU and LOI mutants are trivial

5. Exclude CDL, COD, and VDL when using ODL, because ODL subsumes them

5.4.5 Re-evaluate the Effectiveness

Based on the evaluation results, Section 5.4.4 provides recommendations to eliminate the redundancy among Android mutation operators. However, it is not clear whether the effectiveness of Android mutation testing still holds after removing and modifying redundant mutation operators. Due to the high computational cost of Android mutation testing, re-conducting the whole effectiveness evaluation in Section 5.3 may take several months, which is very time-consuming. Thus, this re-evaluation selected one subject app for spot-checking.

According to the recommendations in Section 5.4.4, I updated the implementation in muDroid, by excluding AODU, AOIU, and LOI, excluding CDL, COD, and VDL when using ODL, excluding BWD when using BWS, and improving the implementation of MDL.

I took Tipster as the subject app for the re-evaluation. Originally, Tipster generated 327 muJava mutants and 130 Android mutants. After removing and modifying redundant mutation operators, Tipster generated 259 muJava mutants and 125 Android mutants, with an overall 16% reduction in terms of the total number of the mutants. After that, a new set of mutation adequate tests was designed. Originally, Tipster had 64 crowdsourced faults, in which 51 were detected by the old mutation adequate test set. After re-conducting the evaluation, the newly designed mutation adequate test set using fewer and less redundant mutants found the same 51 crowdsourced faults in Tipster. Therefore, it is concluded that removing and modifying redundant mutation operators in this research did not impact the effectiveness of Android mutation testing.
5.4.6 Threats to Validity

Similar to most experiments in software engineering, this empirical evaluation has several threats to validity, which could potentially impact the experimental results.

**Internal validity:** In this experiment, only one set of Android mutation-adequate tests was designed for each type of mutants. The results of redundancy scores may differ for different Android mutation-adequate tests, creating a potential threat to internal validity.

Also, in this experimental study, all the equivalent mutants were identified by hand. Manual work could introduce human errors into the artifacts that may affect the final results.

**External validity:** Like in all software engineering experiments, I cannot guarantee that the selected subjects are representative. Using different subject apps, the results and redundancy scores may differ from the results and the redundancy scores in this study. All the Android apps in this study were previously used by other researchers in Android testing research.

**Construct validity:** The implementation of muDroid and the associated mutation operators may include software faults. In this study, muDroid and the experimental environment were constantly tested to make sure they work correctly.
Chapter 6: Conclusions and Future Work

This chapter summarizes the studies conducted in this research, revisits the research questions and the findings, and draws conclusions (Section 6.1). In addition, the chapter lists the contributions of this research (Section 6.2.1), and suggests some future research directions (Section 6.3).

6.1 Conclusions

Android mobile devices and Android apps dominate the global market in terms of the numbers of users, developers, devices, and apps. However, this volume makes the quality problem of Android apps much worse. Severe software failures are frequently observed in many Android apps, such as runtime crashes, incorrect behaviors, and security vulnerabilities. Thus, we desperately need more sophisticated and effective testing.

To make this more difficult, Android apps include new programming features and structures never seen in traditional software before. These unique characteristics introduce new types of software faults into Android apps, but existing software testing techniques and simple testing coverage criteria cannot sufficiently test Android apps or detect these new types of software faults.

New software testing techniques specific for Android apps are being developed, but prior to this research, we did not have effective techniques to evaluate these techniques, or to ensure a reasonable number of effective tests.

This research investigated the programming framework, unique programming features, and novel characteristics of Android apps, and developed Android mutation testing, a more sophisticated testing strategy than current practice. Android mutation testing not only designs effective tests for Android apps, but also supplies an effective evaluation criterion.
for assessing other Android app testing techniques. Redundant or ineffective Android test cases can be filtered out, and ultimately, the ability to deliver quality Android apps can be improved.

The following hypothesis has been validated with eight research questions across three experimental studies.

**Research Hypothesis:**

Mutation testing of Android apps can reveal more faults than existing testing techniques can.

The first experimental study investigated the feasibility of applying mutation analysis to testing Android apps. It verified whether Android mutation testing can be used to evaluate test cases designed with other testing criteria.

- **RQ1:** Is it feasible to test real-world Android apps with mutation analysis?
  
  - muDroid successfully generated 3,275 Java traditional method-level mutants, and 1,706 Android mutants for 8 real-world open source Android apps, and executed pre-designed 100% statement coverage test sets on these mutants.
  
  - Different types of devices, Android emulators and real smartphones, with different Virtual Machines, Dalvik and ART, were assessed in the study. The results show that Android mutation testing can feasibly test real-world Android apps.

- **RQ2:** How effective can test cases designed with traditional testing criteria be in killing mutants generated by Android mutation testing?
  
  - 100% statement coverage tests were evaluated in this study. They missed around 39% of Java traditional method-level mutants, and 30% of Android mutants. Given that every mutant can be considered to represent one or more software faults, 100% statement coverage tests were found to be not very effective.

After exploring the applicability of Android mutation testing, the second experiment
investigated the fault detection effectiveness of Android mutation testing, and compared it with four other Android app testing tools: Monkey [8], Dynodroid [115], PUMA [81], and A³E [47]. In addition, to make this study more comprehensive, this evaluation uses naturally occurring faults as well as crowdsourced faults introduced by experienced Android developers.

• **RQ3:** How effective is Android mutation analysis in testing Android apps? Specifically, how many faults can be detected by mutation-generated tests?

  – In this study, overall, Android mutation-adequate tests detected 18 of 25 naturally occurring faults, and 360 of 437 crowdsourced faults.

• **RQ4:** 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?

  – Of the 25 naturally occurring faults, Dynodroid detected seven, Monkey detected six, PUMA found three, and A³E discovered only one. Of the 437 crowdsourced faults, Dynodroid detected 138, Monkey found 130, PUMA detected 121, and A³E discovered 79.

  – According to the experiment results, for both groups of faults, Android mutation-adequate tests found more faults than the four other Android testing techniques at a statistically significant level. This is not surprising, 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.

• **RQ5:** Is there any difference between using naturally occurring faults and using crowdsourced faults in empirical evaluations?

  – 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, it is not possible to conclude that either population of faults led to different results.

After evaluating the effectiveness of Android mutation testing, the third experiment investigated the possibility of reducing the cost of Android mutation testing by searching for redundant mutation operators.

- **RQ6:** How many mutants of one particular type can be killed by tests created to kill another type of mutants?
  
  Overall, the average redundancy scores ranged from 0% to 100%, which means that certain mutation operators were totally redundant and can be excluded to save costs without downgrading the effectiveness of Android mutation testing, including Unary Arithmetic Operator Deletion (AODU), Unary Arithmetic Operator Insertion (AOIU), and Logical Operator Insertion (LOI).

- **RQ7:** Which types of mutants are less likely to be killed by tests created to kill other types of mutants?
  
  Some Android mutation operators are very hard to kill. Less than 7% of Fail on Back (FOB) mutants were killed. The Back button interrupts the usual execution flow, but it is usually not on the happy path from the perspective of software design. The TextView Deletion (TVD) mutation operator is another type that was very hard to kill in the experiment (less than 1%). Many TextView widgets do not associate with any user events, nor require any event handlers from the implementation of the app, so they are very likely to be overlooked by tests. On average, only 2.5% of the Orientation Lock (ORL) mutants was killed by other types of tests.

- **RQ8:** Are any Android mutation operators redundant enough to be excluded, or can any be improved? In particular, can the mutants generated from this mutation operator always be killed by tests created to kill another type of mutant?
Unary Arithmetic Operator Deletion (AODU) has the highest average redundancy scores among all the muJava and Android mutation operators. 16 mutation adequate test sets killed all AODU mutants, so AODU is redundant and should be excluded. Activity Lifecycle Method Deletion (MDL) has very high redundancy scores with respect to tests designed for killing other types of mutants, because it deletes the definition and initialization of GUI structures in `onCreate()` method, which leads to trivial mutants. A new implementation was suggested in Section 5.4.4. Unary Arithmetic Operator Insertion (AOIU) and Logical Operator Insertion (LOI) also create many trivial mutants, as they mutate the resource IDs of GUI widgets in Android apps, resulting in crashes right after launching. All the Button Widget Deletion (BWD) mutants were killed by Button Widget Switch (BWS) tests, because BWS subsumes BWD. ODL also subsumes CDL, COD, and VDL. These three can also be excluded if ODL is used in Android mutation testing.

In conclusion, the results validated the hypothesis and confirmed that Android mutation testing not only designs effective tests for Android apps, but also supplies an effective evaluation criterion for assessing other Android apps test selection techniques.

6.2 Intellectual Merits

This section summarizes the research contributions of this dissertation, including the research contributions (Section 6.2.1) and the scientific impacts (Section 6.2.2).

6.2.1 Research Contributions

This research developed an effective technique for testing Android apps using mutation analysis, by which testers can design powerful tests for Android apps, or evaluate the effectiveness of a pre-existing test set. Ultimately, our ability to deliver quality Android apps is improved through stronger testing. Particularly, this research makes the following
major contributions:

1. **A fault model for Android apps**
   This research documented common faults in Android apps, as a fault model, by mining open source project repositories and investigating the programming framework, unique features, and novel characteristics of Android apps.

2. **A set of Android mutation operators**
   This research designed 17 novel Android mutation operators for Android apps. These mutation operators address the common faults as documented in my fault model.

3. **An Android mutation testing tool**
   I implemented an Android mutation testing tool, muDroid, based on the Android mutation operators. muDroid is fully compatible with the Android operating system, able to install compiled mutants and execute tests on both Android emulators and real devices, and store mutation execution results.

4. **A repository of Android apps with naturally occurring faults and crowdsourced faults**
   This research delivers a repository of Android apps, which includes hundreds of naturally occurring faults and crowdsourced faults our study used. The repository can serve as a benchmark to assess Android testing techniques, support software engineering experiments for Android apps, and provide research subjects for Android program analysis.

5. **Experimentally evaluated the effectiveness of Android mutation testing**
   This research experimentally evaluated the feasibility and the effectiveness of Android mutation testing. The results showed that Android mutation testing is very effective at detecting naturally occurring faults and crowdsourced faults.

6. **Experimentally found that statement coverage does not provide effective tests**
With the mutation scores of 61.4% for muJava mutants and 70.5% for Android mutants, this research found that statement coverage cannot provide effective tests for Android apps.

7. Experimentally determined that four Android app testing tools are not effective at detecting faults
This research experimentally evaluated and then compared the fault detection effectiveness of four Android app testing tools, both state-of-the-art and state-of-the-practice. The results showed that they are not effective at detecting naturally occurring faults and crowdsourced faults.

8. Experimentally analyzed the redundancy in Android mutation testing
This research experimentally analyzed the redundancy in Android mutation testing, and provided recommendation for reducing the cost of Android mutation testing.

6.2.2 Impacts
This dissertation has the following impacts:

1. To the research community
This research is the first attempt to extend mutation analysis to the domain of mobile apps, which initiates a new innovative research area, Android mutation testing. Researchers interested in designing testing techniques for Android apps can use Android mutation testing to compare or evaluate their methodologies.

2. To Android developers, testers, and the mobile app industry
The quality of an app directly impacts the income of its developers and testers. With the Android mutation testing tool, muDroid, developers and testers of Android apps can design tests that are more effective at detecting software faults. This research can help mobile app vendors deliver higher quality apps, attract more customers, and increase revenue.
3. To the general public

This research can help users of mobile apps experience fewer software failures than before, bringing higher user satisfaction.

6.2.3 Papers

This chapter lists the papers that are based on this dissertation topic and other papers I have co-authored in my Ph.D. study period.

1. Papers based on this dissertation:

   (a) **Lin Deng** and Jeff Offutt, Effectively Testing Android Apps with Mutation Analysis, manuscript in preparation.

   (b) **Lin Deng** and Jeff Offutt, An Empirical Study to Identify Redundant Mutation Operators in Android Mutation Testing, manuscript in preparation.


   (e) **Lin Deng**, Nariman Mirzaei, Paul Ammann, and Jeff Offutt, Towards Mutation Analysis of Android Apps, in 10th Workshop on Mutation Analysis (Mutation 2015), April 2015, Graz, Austria.

2. Other co-authored papers in Ph.D. study period:

6.3 Future Research Directions

This research designed Android mutation testing and demonstrated that it is very effective at designing high quality test cases and evaluating pre-existing test cases. The research into
Android mutation testing is not finished and, there are still avenues for future research and improvement. This section suggests four future research directions.

1. **Better mutation operators**

   (a) The experiment in Section 5.3 identified a possible common fault across several subjects. Many apps designed a “settings” or “preferences” menu to let users configure the apps, but failed to properly save the modified settings after the user changed them, which leads to other incorrect behaviors of the app. Additional mutation operators could be defined to encourage testers to design tests to ensure the settings menu works correctly.

   (b) The implementation of several mutation operators can be improved. The experiment in Section 5.4.4 provided several recommendations regarding reducing the costs of Android mutation testing. For example, improving the algorithm that implements MDL could not only help testers to verify that Activity components behave correctly when switching states, but also generate fewer trivial mutants. Also, muDroid could automatically disable mutation operators subsumed by others. The experiment in Section 5.2 also identified a way to improve the implementation of FON to generate fewer equivalent mutants.

2. **Better tests**

   (a) Many Android apps heavily employ rich GUI components and multimedia representations, such as audio inputs, outputs, and graphics. The experiment described in Section 5.3 showed that when these types of multimedia elements contained faults, it is very difficult to detect them, which becomes a much more general observability problem with test oracles. Consequently, existing strategies for designing test oracle are not as effective as with traditional software. Researchers have designed graphical and audio test oracles in other software domains. muDroid could be used to carry out an empirical study to evaluate
and compare the effectiveness of Android graphical and audio test oracles with existing test oracle strategies.

(b) The current implementation of Android mutation testing supports tests written with several Android automation testing frameworks, such as Robotium [29], Espresso [15], and Selendroid [30]. However, muDroid is not able to support tests for inter-app user events. Certain faults in Android apps may only be revealed when one app calls an Intent in another app, which the tests designed in this research did not support. Adding support for inter-app tests could be very promising.

3. A better tool

(a) The experiment in Section 5.2.1 mentioned that Android mutation testing was designed for native Android apps traditionally running on the Android operating systems with the features supported by Android SDK libraries. However, there are other types of Android apps developed with different methodologies. For example, hybrid apps are implemented with elements of web applications. To improve the applicability of Android mutation testing, it would help to add the support for these types of Android apps by leveraging web mutation testing.

(b) Android mutation testing is still only semi-automated. Even though mutant generation and test execution are automated, test generation and equivalent mutant identification require manual work. Therefore, a promising future research direction is to incorporate new techniques, such as machine learning, to further help generate tests automatically and identify equivalent mutants, and ultimately make it fully automated.

4. Additional empirical studies

(a) Many Android developers release their test suites along with their program source code on open source repositories such as GitHub. These tests are often randomly
generated or created with an ad-hoc process raising the question whether they are effective at detecting software faults? Using Android mutation testing, we could conduct a large scale evaluation against test suites from open source repositories.

(b) Recent research of minimal mutation analysis and dominator mutation score [95, 97] identified that traditional mutation score is inflated during the process of mutation analysis and cannot serve as an ideal measurement for assessing the effectiveness of tests or evaluating the test completeness. Since a very strong and rich test set is necessary to minimal mutation analysis and dominator mutation score, this research did not include them into Android mutation testing, due to the expensive cost of Android mutation testing, in terms of computational time and effort. For the future work, it would be very promising to use minimal mutation analysis and dominator mutation score to check whether the conclusions in this research still hold or not.

6.4 Industrial Application

The ultimate goal of Android mutation testing is to improve our ability to deliver higher quality Android apps. Thus, to help Android developers and testers using the technique designed in this dissertation in their Android app development, I recommend the following future industrial and research extensions.

muDroid was developed to execute from a command line, and every step of Android mutation analysis offers a list of APIs. These development measures provide necessary support for future extensions and integrations. For example, an IDE plugin could be developed using the APIs of muDroid to integrate muDroid to Android development IDEs, such as Android Studio. A cloud server based Android app testing environment could be developed using the APIs of muDroid, so that developers could testing their Android apps on cloud and significantly save the execution cost.

Test automation is a necessary step for industrial application. Researchers always want to provide developers and testers better automated testing techniques to save their time
and cost in software development. Particularly for Android mutation testing, I interpret test automation in three perspectives: automated input generation, automated test oracle generation, and fewer equivalent mutants.

Automated input generation helps testers design better test inputs with less effort. Recent research has incorporated deep learning techniques to produce test inputs [111]. Including automated input generation in Android mutation testing is definitely a necessary and feasible step.

Automated test oracle generation is a promising but challenging aspect in test automation. This dissertation suggested graphical, audio, and multimedia test oracles in Android app testing. Leveraging machine learning techniques, I believe integrating automated test oracle generation into Android mutation testing is feasible.

Like mutation testing in other software domains, equivalent mutants contribute to the cost of Android mutation testing. It is undesirable to make developers and testers manually identify all equivalent mutants, even though they are the people who understand their apps thoroughly. Thus, generating fewer equivalent mutants is a necessary step for test automation of Android mutation testing. This could be achieved by improving the design of mutation operators through a subsumption and redundancy analysis on equivalent Android mutants. Making the technique detect equivalent mutants would also help save the cost of Android mutation testing.

In summary, I would like to devote more efforts to the technical and research extensions for Android mutation testing, to shorten the distance from research findings to industrial applications, to ultimately improve our ability to deliver higher quality Android apps.
Appendix A: Acronyms

**ACRT** Android Capture and Replay Testing Tool

**ADT** Android Developer Tools

**AODS** Arithmetic Operator Deletion, Short-cut

**AODU** Arithmetic Operator Deletion, Unary

**AOIS** Arithmetic Operator Insertion, Short-cut

**AOIU** Arithmetic Operator Insertion, Unary

**AORB** Arithmetic Operator Replacement, Binary

**AORS** Arithmetic Operator Replacement, Short-cut

**AORU** Arithmetic Operator Replacement, Unary

**APD** Activity Permission Deletion

**APIs** Application Programming Interfaces

**APK** Android Application Package

**ART** Android Runtime

**ASRS** Assignment Operator Replacement, Short-cut

**BWD** Button Widget Deletion

**BWS** Button Widget Switch

**CDL** Constant DeLetion

**COD** Conditional Operator Deletion

**COI** Conditional Operator Insertion
COR  Conditional Operator Replacement

DOM  Document Object Model

ECR  OnClick Event Replacement

EDC  Evans Data Corporation

ELOC  Executable Lines of Code

ETR  OnTouch Event Replacement

FOB  Fail on Back

FON  Fail on Null

GPS  Global Positioning System

GUI  Graphical User Interface

HAXM  Hardware Accelerated Execution Manager

IDC  International Data Corporation

IP  Internet Protocol

IPR  Intent Payload Replacement

ITR  Intent Target Replacement

JPF  Java PathFinder

LCM  Location Modification

LOD  Logical Operator Deletion

LOI  Logical Operator Insertion

LOR  Logical Operator Replacement

MDL  Activity Lifecycle Method Deletion
NLP  Natural Language Processing
NOS  Null Test Oracle Strategy
NPE  Null Pointer Exception
ODL  Operator Deletion
ORL  Orientation Lock
RERAN  REcord and Replay for ANdroid
ROR  Relational Operator Replacement
SDL  Statement Deletion Mutation Operator
SDL  Statement Deletion
SLOC  Source Lines of Code
SMDL  Service Lifecycle Method Deletion
SOR  Shift Operator Replacement
SPAG-C  SmartPhone Automated GUI Testing tool with Camera
SURF  Speeded Up Robust Features
TVD  TextView Deletion
TWD  EditText Widget Deletion
VDL  Variable Deletion
VM  Virtual Machine
WCD  Wi-Fi Connection Disabling
**WRD** WakeLock Release Deletion

**XML** eXtensible Markup Language
Bibliography
Bibliography


[53] Graeme Burton. RBS claims to have found and fixed payments IT glitch that affected 600,000. Online, June 2015. http://www.computing.co.uk/2414023, last access June 2017.


[112] Localytics. App retention improves - apps used only once declines to 20%. Online, June 2014. [http://info.localytics.com/blog/app-retention-improves](http://info.localytics.com/blog/app-retention-improves), last access June 2017.


Curriculum Vitae

Lin Deng is a Ph.D. candidate in the Department of Computer Science of Volgenau School of Engineering at George Mason University. He received his M.S. in Computer and Information Sciences from Gannon University in Pennsylvania in 2011. Before that, he received a B.E. in Computer Science from Renmin University of China in 2005. His research interests include software testing, mobile application development, and usable security. His advisor is Dr. Jeff Offutt.