Automated Test Generation for Android Devices

Michael Goff
Computer Science Department
North Carolina State University, USA
Magoff2@ncsu.edu

Laurel Timko
Computer Science Department
North Carolina State University, USA
Latimko@ncsu.edu

Peng Wang
Computer Science Department
North Carolina State University, USA
Pwang13@ncsu.edu

ABSTRACT

Android applications have become a part of our daily lives from communication to banking to e-commerce. As a result, the need to test these applications has essential. In this paper, we will discuss the different types of issues that are commonly found in Android applications, such as Event and Activity Bugs, along with the techniques that are used to catch each issue. The most prevalent techniques used are random-based, model-based, and systematic-based testing. From there, we will examine several open-source testing applications that have been analyzed by various academics for strengths and weaknesses, identify what kinds of bugs they are able to identify, and how they compare to similar tools.

CCS Concepts

• Android/ Mobile Application Development • Automated Testing • Software Engineering

Keywords

Android Operating System; Mobile Application; Automated Test Generation

1. INTRODUCTION

Since its initial release in 2008 [17], the Android operating system, developed by Google, has been gaining momentum and now accounts for over 50% of smartphone sales since 2011 [1]. In fact, Android is now the leading smartphone platform in the world, surpassing Apple's iOS [7]. However, while the Android application marketplace has exploded to accommodate the increasing demand, the tools available to test these applications have been slow to catch up for a number of reasons.

Android applications utilize Java, but cannot be tested using the traditional Java automated testing tools, as these applications are often compiled into Dalvik bytecode, which is inherently different from Java bytecode used in conventional desktop application [1, 8]. Newer versions of Android have since switched away from Dalvik to a new system called Android Runtime (ART). Released after many of these previous studies and tools, ART has some notable advantages over Dalvik including ahead of time compilation, improved garbage collection and a plethora of debugging improvements comprising of more robust exceptions and crash reports [14]. These new features may lead to significant improvements in the many tools discussed throughout the paper. However, Android devices also come from a number of manufacturers and support numerous older OS versions; this makes the Android applications susceptible to platform and version incompatibilities [8].

Another challenge of testing mobile applications as a whole is generating sufficient test cases that can test the wide breadth of possible user inputs and the context of the environment, such as the other applications running in the background on the device [4]. These heavily GUI-based applications are structured around the concept of activities and services, which enable user events to complete a task [1]. Identifying and testing all possible event sequences becomes increasingly difficult for larger, more complex applications, especially ones where security and privacy become crucial [3]. As a result, Android testing tends to be a manual process, which is both slow and expensive.

The number of available automated testing tools has seen substantial growth over the last few years, with each iteration improving and concentrating on a new detail to software testing [5]. In this paper, we look at the practices and techniques being utilized in the development of such tools relating to Android application testing. We are specifically looking at the Android platform as it has undergone immense academic research, due to it's open-source nature and the resulting open-source nature of many of the testing tools developed around it [8].

2. Bug Identification

The following sections detail the various classifications of bugs that Android applications may be plagued by and the common techniques that are used to identify them.

2.1 Classification of Bugs

There are many different types of bugs that may exist within an Android application. The following section describes just a few of the more easily targeted types.

2.1.1 Poor Responsiveness

Today's mobile applications are often dependent upon speed, as users rely on instant gratification. When it takes an Android "application more than 200 ms to respond to a user event, it is perceived to be sluggish and unresponsive," falling into the category of having poor responsiveness [7]. There are varying degrees of unresponsiveness with the worst resulting in an application crash while displaying an "Application Not Responding" error.

2.1.2 Event Bugs

Events are actions performed by a user on the application's user interface; these events input data to the application through motion, gestures and taps [6]. Event bugs are defined as bugs that occur when an event causes the application to perform an incorrect action [2]; these bugs can also cause the entire application to crash. Examples of such bugs can be null pointer exceptions or array index out of bounds errors, where the developer failed to handle a situation where the application entered into an improper state. Examples are also seen in Figure 2 below.

2.1.3 Activity Bugs

An activity is a screen within an application that allows a user to interface and send input to the application [9]. Activity bugs are errors that result from the incorrect use of the activity protocol or violating the activity life cycle [2]. In other words, a developer may improperly instantiate and manipulate an activity due to a lack of experience and knowledge of the activity lifecycle. This lifecycle is

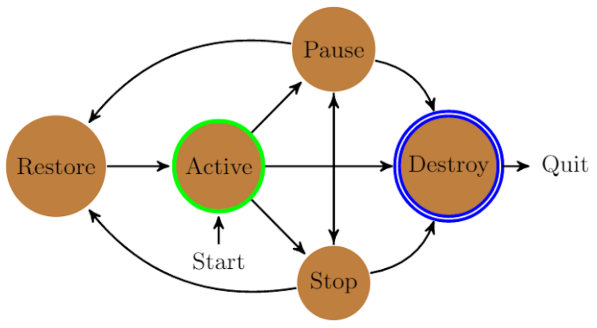


Figure 1: State machine visualization of an Android Activity [2]

clearly defined by the activity state machine, seen simplified in Figure 1, and is often implemented by test generation tools to determine when it has been violated. An example of what an Activity Bug may look like is seen in Figure 2.

2.1.4 Other Bug Types

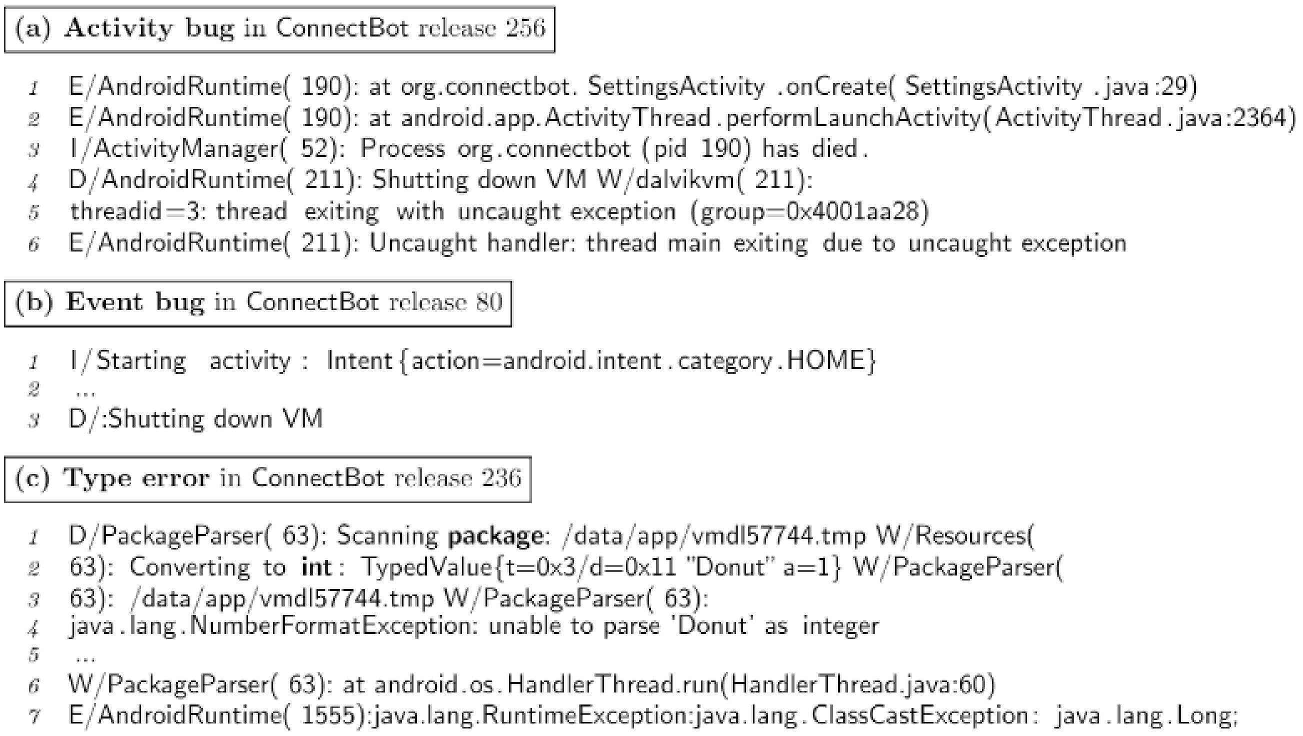
Another classification of bugs, Dynamic type bugs, occur during runtime as a result of type exceptions [2]. Errors like these can result from improper integer or floating point conversions or parses. This type of error tends to be more easily detected and reported, usually because it results in the application crashing and an appropriate message will appear in the logs for easy pattern matching. An example of a type bug is seen in Figure 2.

Concurrency errors are another source of errors for Android applications. Concurrency errors are issues that arise during the interaction of different threads or processes [2]. These threads could be from different applications, running in the background on the device. While some of the available tools are able to identify a handful of these issues, this particular kind of error is still able to pass into production undetected. Authors Hu and Neamtiu are working on a new approach using state machines that will allow a testing application to recognize and report on these errors; this future work will involve modelling the state machine based on known I/O and concurrent points, to identify errors in the logs [2].

API calls make up yet another potential source of challenges for automated testing in Android applications. This can be broken down into network access calls, flash storage access calls, database calls, and bitmap processing [7].

3. Testing Techniques

There are a variety of testing techniques that may be used for automated testing tools. Each technique has their own advantages and costs associated with them. The major techniques are random, systematic and model based testing. Random testing is typically a quick solution but risks not covering everything. Systematic testing is great for testing more inputs but falters when it comes to scalability. Finally, Model based testing offers a closer emulation



(a) Activity bug in ConnectBot release 256

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

(b) Event bug in ConnectBot release 80

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

(c) Type error in ConnectBot release 236

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

Figure 2: Examples of Activity, Event, and Dynamic Type Errors [2]

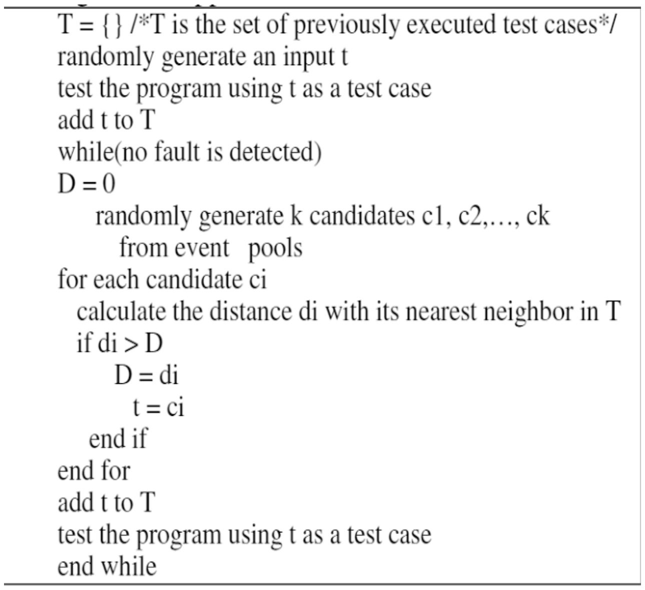
of the GUI but may miss certain classifications of bugs and is only ever as good as the model allows.

3.1 Random

Until more recently, most testing methods revolved around an element of randomness. Test generation tools would randomly generate events to be performed on the GUI. This is ideal for stress testing as a large number of tests and test inputs can be generated at once [8]. However, this method of testing does not yield the best code coverages as not all possible branches may be tested thoroughly; it is harder to generate specific input using randomly generated events and more complex use cases are likely to be missed. Many testing tools still incorporate randomly generated events, but now they have expanded to use other testing strategies and techniques described below.

3.1.1 *Adaptive Random Testing*

Adaptive Random Testing (ART) is a method of testing that utilizes a statistical approach to measure and categorize failure patterns, so that test cases may be more evenly spread out over the complete set of inputs [10]. This testing technique stems from the desire to produce high quality tests scripts through the direct recording and replaying of tests. In 2010, Liu et al. modified the original algorithm proposed by Chen et al. in 2004, such that the algorithm randomly selects events from within a randomly generated event pool, in attempt to increase test diversity [4, 10]. This new script is seen below in Figure 3. The authors do not compare how their modified adaptive random testing algorithm compares to the original algorithm, as they were focused on how the algorithm improved from traditional random testing, so this cannot be definitively labeled an improvement. ART is successful in evenly spreading out the test cases, as it requires far fewer test cases compared to random testing to find the first fault in less time. This is evident in Figures 4 and 5.



```
T = { } /*T is the set of previously executed test cases*/
randomly generate an input t
test the program using t as a test case
add t to T
while(no fault is detected)
D = 0
      randomly generate k candidates c1, c2,…, ck
          from event   pools
for each candidate ci
   calculate the distance di with its nearest neighbor in T
   if di > D
         D = di
            t = ci
     end if
end for
add t to T
test the program using t as a test case
end while
```

Figure 3: Adaptive Random Testing Algorithm Used by Liu et al. [4]

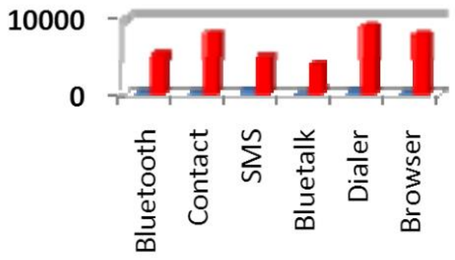


Figure 4: Results Comparison of the Tests Cases to First Fault (F-Measure) between ART and Random Testing

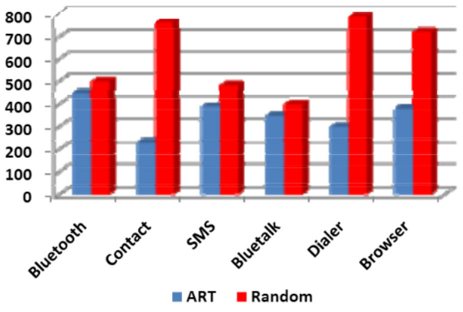


Figure 5: Results Comparison of the Time to First Fault between ART and Random Testing

3.2 Systematic Testing

As mentioned above, a weakness of random testing is the inability to test specific inputs. In contrast, systematic testing is able to improve test coverage by generating tests for unique inputs. To do so, techniques such as symbolic execution and other algorithms are employed. However, these techniques often in result in the tools lacking scalability [8].

3.2.1 *Concolic Execution*

Concolic execution is a hybrid process that combines both symbolic and concrete execution to execute and test a program. With this technique, concrete execution dictates the symbolic execution, as the concrete set of paths provided from the concrete execution are tested with the symbolic inputs provided by symbolic execution. In 2013, Jensen et. al developed a new approach inspired by symbolic execution and model-based testing with the intent of generating complex tests sequences. Jensen et. al utilized concolic execution in combination with event sequence builders in order to test long, complicated sequences of events with different parameter inputs. This specific implementation, named Collider, first runs concolic execution on individual event handlers of the application and then uses the resulting data to make conclusions about the relationship between events and reason through event sequences backwards from the GUI model [5]. The script used by the authors to generate these event sequences is seen below in Figure 6. This concolic approach is very similar to a technique used by an application discussed later in this paper called ACTEve, but Collider differs in its singular particular targets of the individual event handlers, rather than over the entire application from the entry point. As such, this approach is able to scale for larger event sequences to maximize the code coverage of tests, while being able to consistently reproduce the same errors [5, 13]. The results

Table 1: Experimental Results of GUI Based Test Amplification [7]

Application	Version	Activities	Classes	Test Cases	Network		Flash		Database		Bitmap	
					All	Fail	All	Fail	All	Fail	All	Fail
APV	r131	4	56	13	0	0	5	1	4	1	0	0
astrid	cb66457	11	481	18	2	0	11	0	62	3	0	0
ConnectBot	e63ffdd	9	301	14	6	1	5	1	23	14	0	0
FBReader	a53ed81	22	757	17	2	0	22	4	43	8	2	1
KeePassDroid	085f2d8	7	126	21	0	0	7	0	14	6	0	0
K9	v0.114	15	418	14	20	0	25	1	41	9	3	0
VLC	dd3d61f	8	176	18	1	0	8	1	22	3	8	1
VuDroid	r51	3	67	11	0	0	2	2	0	0	4	4

indicate that compared to sample applications like Monkey and other simple crawler tools, Collider is able to reach more of the manually identified challenging targets.

The authors were good to analyze their findings and look at threats to validity and room for improvement. They found that their prototype limits the number of reachable challenging targets because it only supports numeric and boolean values, as opposed to strings and objects, which they labeled as future work. Jensen et. al identify their selection of real-world applications as a threat to the validity of their findings, as it could impact the scalability of their benchmarks. The authors hypothesize that more, if not all, of the predicted challenging sequences could be reached if the algorithm is improved, specifically the symbolic constraint solver component of the algorithm [5]. The authors also compare the performance of their technique to that of other tools, such as Dynodroid, which will be discussed later in this paper; Jensen et. al acknowledge that Dynodroid, a random execution based tool, may be considered faster than this symbolic method with its fewer test inputs, but it will not reach the vast number of challenging targets that Collider did in this scenario [5, 6].

```
function SEQUENCESEARCH(target, summaries, model)
    worklist = INITIALIZE(target, summaries, model)
    while worklist is not empty do
        partialSequence = DEQUEUE(worklist)
        extendedPartialSequences = empty list of sequences
        for anchor in ANCHORS(partialSequence, summaries, model) do
            for path in PATHS(anchor, partialSequence, summaries, model) do
                newPartialSequence = COMBINE(anchor, path, partialSequence)
                if ISCOMPLETE(newPartialSequence) then
                    potentialTestCase = EXTRACTTESTCASE(newPartialSequence)
                    if REACHESTARGET(potentialTestCase, target) then
                        return potentialTestCase
                    end if
                end if
                APPEND(extendedPartialSequences, newPartialSequence)
            end for
        end for
        ENQUEUE(worklist, extendedPartialSequences)
        PRIORITIZE(worklist, extendedPartialSequences)
    end while
    return no test case found
end function
```

Figure 6: Symbolic Event Sequence Generation Pseudo-Script [5]

3.3 Model Based Testing

Similar to systematic testing, some testing tools use a model of the application's GUI, often in the form of a state machine, to generate events and systematically sample the behavior of the application. The state machine uses the events as transitions to activities which represent the various states an application can be in [8]. This enables the discovery of activity and event bugs, described above, in an efficient manner as it removes some of the redundancy that is created when random testing continually runs through the same branch of code with different input. Some classifications of bugs may still be missed with this method of testing, as the tests generated are only as good as the state machine. If a change does not cause the machine to transition to a new state, then the tool may not register that path and continue testing it [8].

3.3.1 Adaptive Random Testing

In response to the absence of a systematic testing strategy for poor responsiveness errors, Yang et. al developed a technique to sequentially amplify and test the various states and transitions of a GUI state machine [7]. The authors utilized the AndroidRipper tool, discussed later in this paper, to generate GUI models for the applications they were evaluating; they commented that the tool did not generate sufficiently comprehensive models, and had to supplement several cases with manual testing. Once the model is constructed from the GUI, a set of test cases is generated with at least one test case for each state and each transition. For this method to be successful, each test is run against the base application and then against an amplified application, as to test the robustness of the application. An amplified application throws additional API calls, like database and network calls, with time delays and exceptions in an attempt to trigger an ANR error.

The authors choose eight sample applications that were open-source applications from several domains such as email and ssh clients, PDF and e-book readers, task and password managers, and multimedia players, as to have a variety of API calls being made [7]. The results of the study are seen below in Table 1.

The authors concluded that their technique was able to detect poor responsiveness errors in the various unique case studies. Yang et al. acknowledged and discussed other numerous related ways to conduct model-based GUI testing, but do not go in depth on these related works as they emphasize functional correctness as opposed to the responsiveness challenge the authors of this paper were addressing. Yang et. al also continued to point out that there is room for improvement and exploration in the areas of automated discovery (both static and dynamic analysis) that is able to manipulate expensive resources and design patterns for preventing responsiveness defects, as opposed to just detecting them [7].

4. Growth of Available Tools

In this section, we will look at the variety of tools that have become available in recent years and how they have improved upon each other.

4.1 Monkey

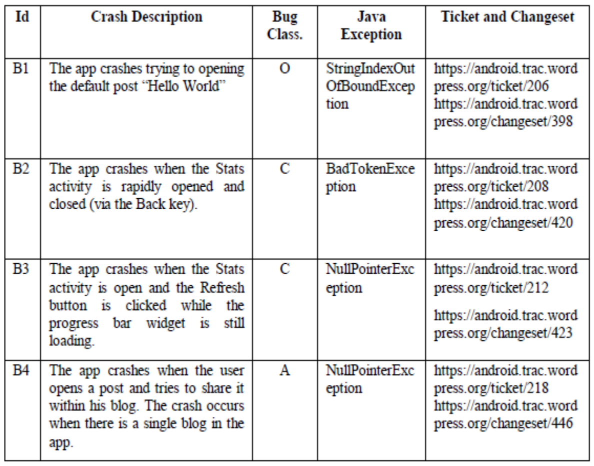
Monkey is a non-commercial command-line tool provided by Android in the Android SDK to test basic user events on an application [1, 9]. This tool allows for stress and crash testing by generating a simulated random set of events, based on pre-configured options, constraints, and frequencies [9]. Monkey is also able to trace log files produced when tests are executed to identify potential bugs based off of known patterns [1]. This tool is able to detect or respond to unhandled exceptions and "Application Not Responding" errors; when these issues are found, "Monkey will stop and report the error," [9]. This is perhaps the most used and cited test generator as it provided in the Android toolkit. It is often used as a baseline for developing tools.

4.2 AndroidRipper

AndroidRipper, also referred to as GUIRipper, is an open source automated testing tool that validates an Android application's functionality by sequentially exploring it's GUI. This application developed, by Amalfitano et. al, relies on the concept of ripping, or reverse engineering, the UI, generating test cases as it goes [1]. As it finds an event that can be executed using the GUI that will result in the model transitioning to a new state, then a test will be created. These states and transitions are managed and tracked in a GUI Tree [1]. This tool is best at catching Event, Activity, and Type errors.

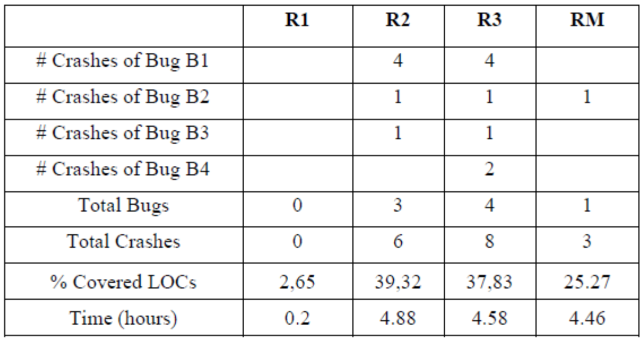
The authors sampled their tool against the open-source Android app "Wordpress for Android," as they considered it desirable for it's active user community and development, along with the well documented issue tracking system employed by the application's developers. The authors also compared their tool to the Android provided testing tool, Monkey, discussed above, as to have a baseline of the number of tests that could be found in the Wordpress application. The results, seen below in Tables 2 and 3, illustrate that AndroidRipper is an improvement upon Monkey, as four new bugs were uncovered with improved line coverage and processing cost, with fewer crashes [1].

Table 2: Results of Types Bugs Found Running AndroidRipper [1]



Id	Crash Description	Bug Class.	Java Exception	Ticket and Changeset
B1	The app crashes trying to opening the default post "Hello World"	O	StringIndexOutOfBoundException	https://android.trac.wordpress.org/ticket/206 https://android.trac.wordpress.org/changeset/398
B2	The app crashes when the Stats activity is rapidly opened and closed (via the Back key).	C	BadTokenException	https://android.trac.wordpress.org/ticket/208 https://android.trac.wordpress.org/changeset/420
B3	The app crashes when the Stats activity is open and the Refresh button is clicked while the progress bar widget is still loading.	C	NullPointerException	https://android.trac.wordpress.org/ticket/212 https://android.trac.wordpress.org/changeset/423
B4	The app crashes when the user opens a post and tries to share it within his blog. The crash occurs when there is a single blog in the app.	A	NullPointerException	https://android.trac.wordpress.org/ticket/218 https://android.trac.wordpress.org/changeset/446

Table 3: Results of Bugs Found Running AndroidRipper Compared to Bugs Found Running Monkey [1]



	R1	R2	R3	RM
# Crashes of Bug B1		4	4	
# Crashes of Bug B2		1	1	1
# Crashes of Bug B3		1	1	
# Crashes of Bug B4			2	
Total Bugs	0	3	4	1
Total Crashes	0	6	8	3
% Covered LOCs	2,65	39,32	37,83	25.27
Time (hours)	0.2	4.88	4.58	4.46

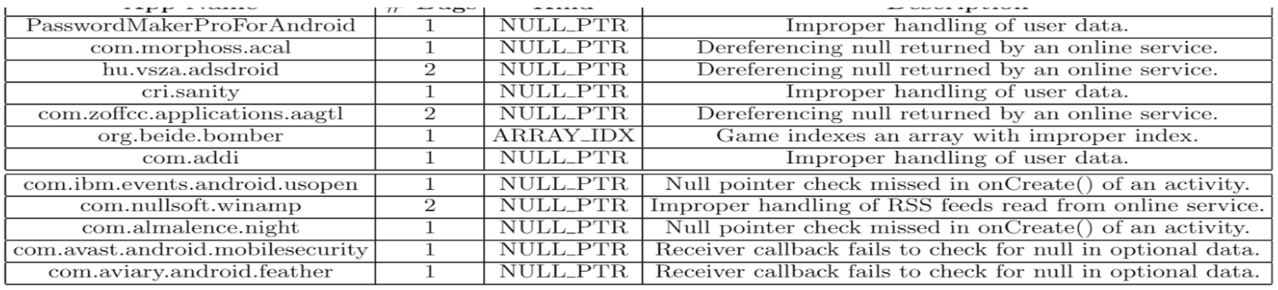
However, this application is not perfect and Amalfitano et. al did not acknowledge any areas of improvement or future work. As discussed previously, Yang et. al used the AndroidRipper tool to reverse engineer GUI models from the applications they were evaluating, but felt that the weaknesses of AndroidRipper required further manual supplementation in some cases. Yang et. al presented alternative approaches that could be used to improve upon or are currently considered, in their present state, improvements upon the AndroidRipper model. Yuan and Memon used the reverse engineering of the GUI as only one half of the development of their model. The other half stems from a test suite of seeded, optimal test cases. These test cases are considered optimal as new cases are generated from and executed on the GUI model until target test coverage is achieved [11]. The only potential disadvantage with this method is that it has yet to be applied to Android based applications.

AndroidRipper is also compared to several other applications, discussed and seen in the figures in the next section. AndroidRipper has the lowest code coverage of all the tools, but this is not without hope. AndroidRipper is able to allow the user look at different views, or snapshots, of the application during testing by supplying their own input; this is referred to as mocking. As such, a better infrastructure could be put in place to utilize this concept of mocking to create more tests over more inputs and increase code coverage [8].

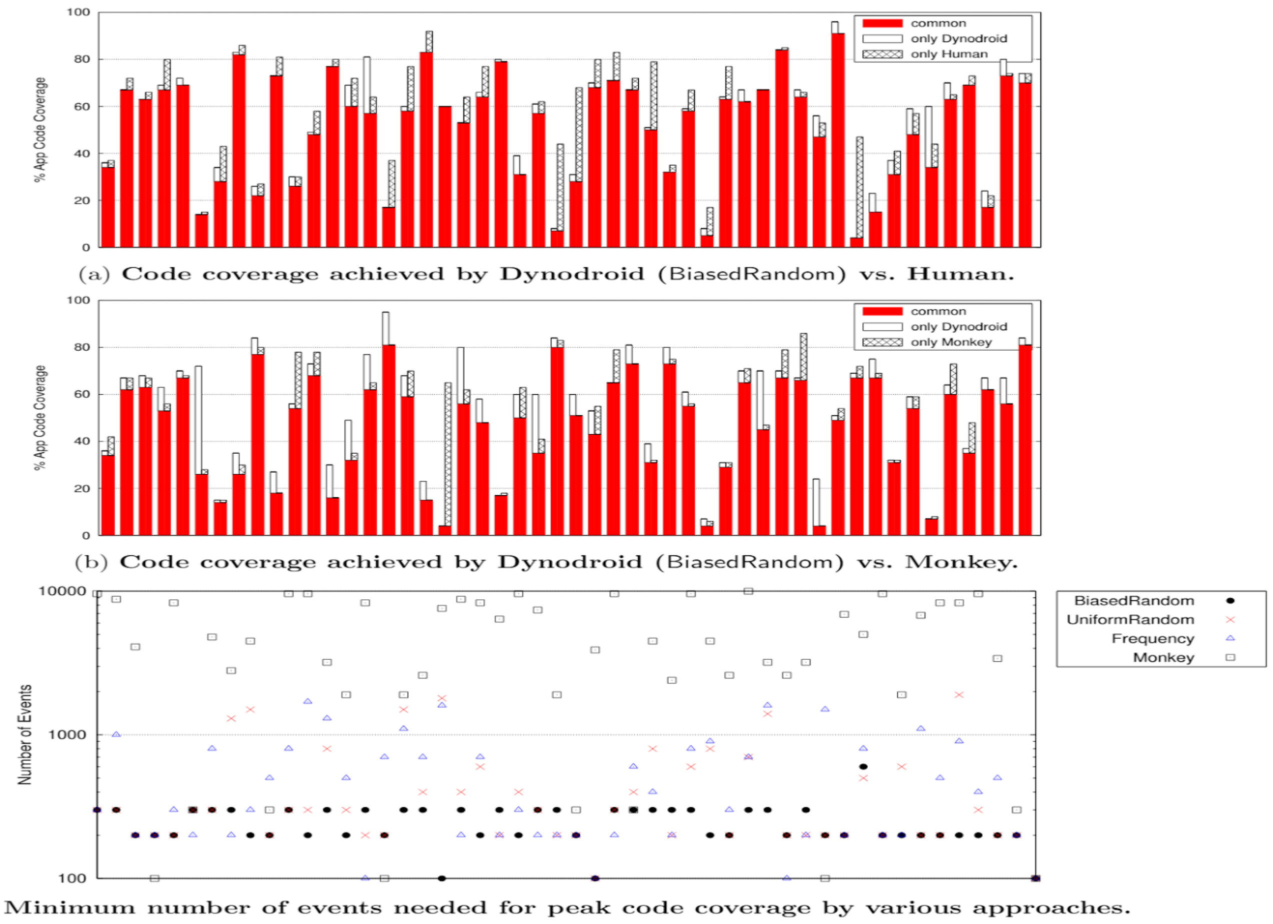
4.3 Dynodroid

Dynodroid is an open source tool, developed by Machiry et. al, that views an a mobile application event-driven program that monitors and interacts with its environment through event sequences. As such, Dynodroid is able to determine when an event occurs, monitor the reaction, and guide to the process to the next event [6]. This is considered a random based testing method as though it follows a sequential processing, the sequence of events that transpire are randomly generated system events. The state machine visualization of Dynodroid is seen below in Figure 8. Dynodroid focuses on the Observe-Select-Execute cycle. The Observe phase is when the application identifies (observes) the events that are relevant and necessary to the current functionality. The Select phase is where a particular event is selected. Finally, in the Execute phase, the selected event is used to trigger a process and cause the application to move to a new state with new potential events [6].

Table 4: Event Classified Bugs Found by Dynodroid [6]



PasswordMakerProForAndroid	1	NULL_PTR	Improper handling of user data.
com.morphoss.acal	1	NULL_PTR	Dereferencing null returned by an online service.
hu.vsza.adsdroid	2	NULL_PTR	Dereferencing null returned by an online service.
cri.sanity	1	NULL_PTR	Improper handling of user data.
com.zoffcc.applications.aagtl	2	NULL_PTR	Dereferencing null returned by an online service.
org.beide.bomber	1	ARRAY_IDX	Game indexes an array with improper index.
com.addi	1	NULL_PTR	Improper handling of user data.
com.ibm.events.android.usopen	1	NULL_PTR	Null pointer check missed in onCreate() of an activity.
com.nullsoft.winamp	2	NULL_PTR	Improper handling of RSS feeds read from online service.
com.almalence.night	1	NULL_PTR	Null pointer check missed in onCreate() of an activity.
com.avast.android.mobilesecurity	1	NULL_PTR	Receiver callback fails to check for null in optional data.
com.aviary.android.feather	1	NULL_PTR	Receiver callback fails to check for null in optional data.



(a) Code coverage achieved by Dynodroid (BiasedRandom) vs. Human.

(b) Code coverage achieved by Dynodroid (BiasedRandom) vs. Monkey.

Minimum number of events needed for peak code coverage by various approaches.

Figure 7: Baseline and New Results of Dynodroid Compared to Monkey and Human Testing [6]

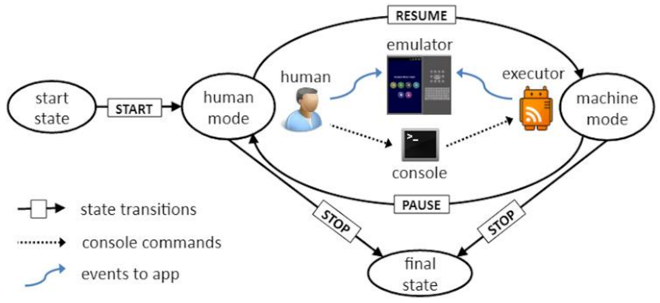


Figure 8: State Machine Visualization of Dynodroid [6]

Dynodroid was run against both the out of the box Monkey tool and against Human developed tests for a baseline comparison. All of these tools were tested consistently against 50 randomly selected open-source mobile applications from different sectors of the Android marketplace. They were all tested on the same hardware, Linux machines with 8GB memory and 3.0GHz processors, and same software version Android's Gingerbread. At the time, this version was the most popular software version, not necessarily the newest or most stable, as it is installed on the most devices. The new results and baseline results are seen below in Figure 7.

Machiry et. al concluded that Dynodroid was successfully able to generate more automated tests with more concise input than both Monkey and Human that exposed new bugs. This method is particularly good at fleshing out Event bugs as seen in the Table 4.

Dynodroid was also evaluated against other random based tools like AndroidRipper/GUIRipper, along with Systematic based tools, in a more extensive third-party study. This study was conducted on a total of 68 apps, most were gathered from the original studies of the applications to assess the validity of the study. The checklist used for this study is seen below in Figure 10. The authors of the study, Choudhary et. al, evaluated all of the applications based on ease of use, android framework compatibility, code coverage achieved, and fault detection ability.

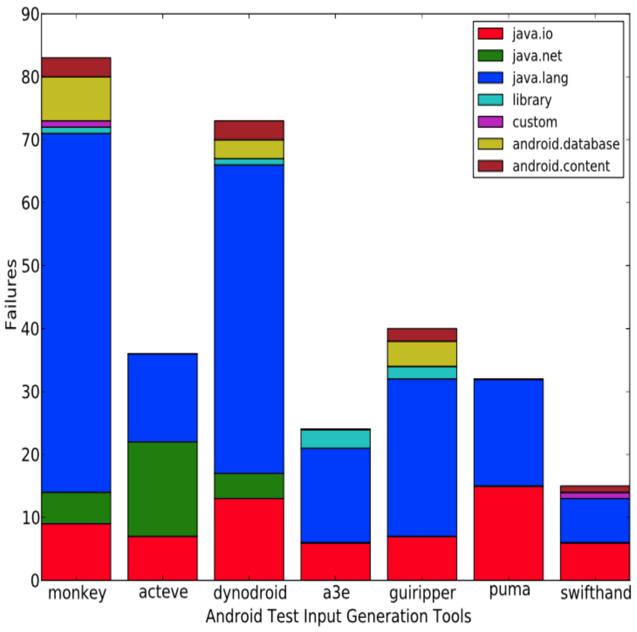
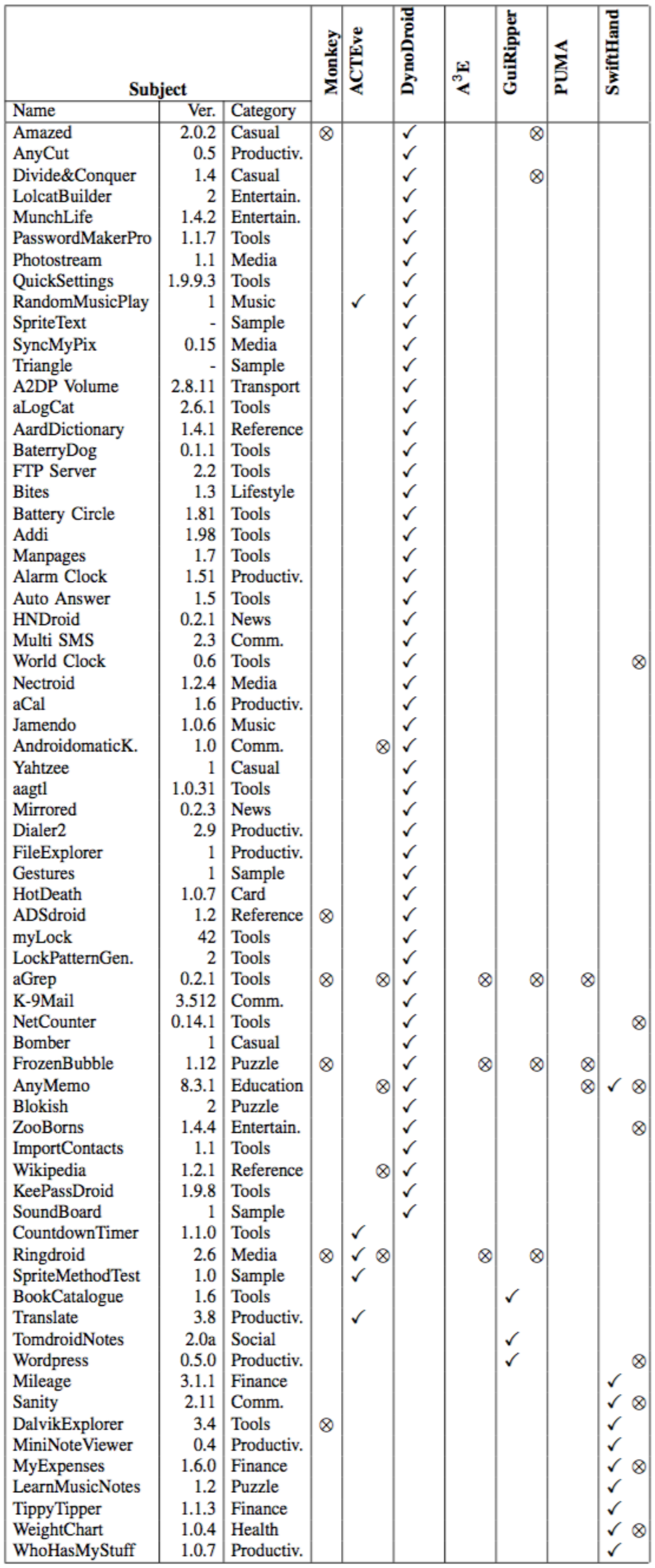


Figure 9: Results of Third Party Study on the Number of Faults Triggered by Tools [8]

The results of this study are seen below in Figures 9 and 11. As compared to the other random based methods, Dynodroid identified more failures in almost every level compared to GUIRipper, and most levels of Monkey, except Android.database and Custom [8]. This would indicate that Observe-Select-Execute pattern that Dynodroid uses is more effective than GUI ripping [1, 6]. Compared to the systematic application ACTEve, discussed in section 4.5, Dynodroid significantly outperforms ACTEve in every category except java.net. This would indicate that the java.net faults, like network API call errors, are more difficult to reach and require more than just randomization. With regard to code coverage, Monkey still has the highest mean code coverage, followed by Dynodroid. This would suggest that random based modeling methods consistently achieve higher code coverage than other methods. This makes sense as tools like Collider and ACTEve stated that they were focused on reaching more complex targets as opposed to increasing code coverage by hitting all of the easy "low-hanging" targets that Dynodroid and GUIRipper can get.



Subject			Monkey	ACTEve	DynoDroid	A^3E	GuiRipper	PUMA	SwiftHand
Name	Ver.	Category							
Amazed	2.0.2	Casual	⊗		✓		⊗		
AnyCut	0.5	Productiv.			✓				
Divide&Conquer	1.4	Casual			✓		⊗		
LolcatBuilder	2	Entertain.			✓				
MunchLife	1.4.2	Entertain.			✓				
PasswordMakerPro	1.1.7	Tools			✓				
Photostream	1.1	Media			✓				
QuickSettings	1.9.9.3	Tools			✓				
RandomMusicPlay	1	Music		✓	✓				
SpriteText	-	Sample			✓				
SyncMyPix	0.15	Media			✓				
Triangle	-	Sample			✓				
A2DP Volume	2.8.11	Transport			✓				
aLogCat	2.6.1	Tools			✓				
AardDictionary	1.4.1	Reference			✓				
BaterryDog	0.1.1	Tools			✓				
FTP Server	2.2	Tools			✓				
Bites	1.3	Lifestyle			✓				
Battery Circle	1.81	Tools			✓				
Addi	1.98	Tools			✓				
Manpages	1.7	Tools			✓				
Alarm Clock	1.51	Productiv.			✓				
Auto Answer	1.5	Tools			✓				
HNDroid	0.2.1	News			✓				
Multi SMS	2.3	Comm.			✓				
World Clock	0.6	Tools			✓				⊗
Nectroid	1.2.4	Media			✓				
aCal	1.6	Productiv.			✓				
Jamendo	1.0.6	Music			✓				
AndroidomaticK.	1.0	Comm.		⊗	✓				
Yahtzee	1	Casual			✓				
aagtl	1.0.31	Tools			✓				
Mirrored	0.2.3	News			✓				
Dialer2	2.9	Productiv.			✓				
FileExplorer	1	Productiv.			✓				
Gestures	1	Sample			✓				
HotDeath	1.0.7	Card			✓				
ADSdroid	1.2	Reference	⊗		✓				
myLock	42	Tools			✓				
LockPatternGen.	2	Tools			✓				
aGrep	0.2.1	Tools	⊗	⊗	✓	⊗	⊗	⊗	
K-9Mail	3.512	Comm.			✓				
NetCounter	0.14.1	Tools			✓				⊗
Bomber	1	Casual			✓				
FrozenBubble	1.12	Puzzle	⊗		✓	⊗	⊗	⊗	
AnyMemo	8.3.1	Education		⊗	✓			⊗	✓ ⊗
Blokish	2	Puzzle			✓				
ZooBorns	1.4.4	Entertain.			✓				⊗
ImportContacts	1.1	Tools			✓				
Wikipedia	1.2.1	Reference		⊗	✓				
KeePassDroid	1.9.8	Tools			✓				
SoundBoard	1	Sample			✓				
CountdownTimer	1.1.0	Tools		✓					
Ringdroid	2.6	Media	⊗	✓ ⊗		⊗	⊗		
SpriteMethodTest	1.0	Sample		✓					
BookCatalogue	1.6	Tools					✓		
Translate	3.8	Productiv.		✓					
TomdroidNotes	2.0a	Social					✓		
Wordpress	0.5.0	Productiv.					✓		⊗
Mileage	3.1.1	Finance							✓
Sanity	2.11	Comm.							✓ ⊗
DalvikExplorer	3.4	Tools	⊗						✓
MiniNoteViewer	0.4	Productiv.							✓
MyExpenses	1.6.0	Finance							✓ ⊗
LearnMusicNotes	1.2	Puzzle							✓
TippyTipper	1.1.3	Finance							✓
WeightChart	1.0.4	Health							✓ ⊗
WhoHasMyStuff	1.0.7	Productiv.							✓

Figure 10: Checklist of Applications sampled in the Cross Tool Study [8]

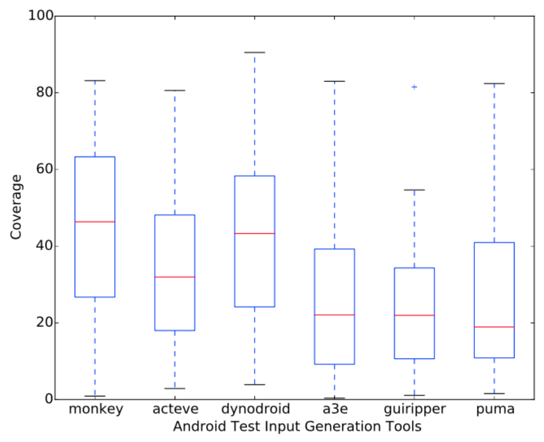
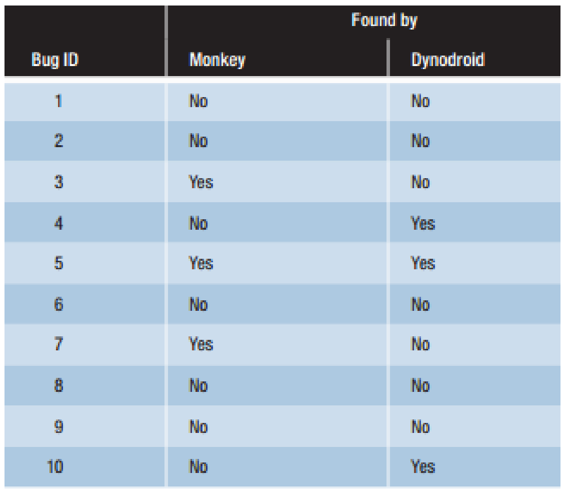


Figure 11: Results of Third Party Study on Statistical Variance Code Coverage Achieved Across 10 Runs by Tools [8]

4.4 MobiGUITAR

Though not stated in the original paper, AndroidRipper authors, Amalfitano et. al, did have future work planned in the form of the tool MobiGUITAR [1, 12]. Released in 2015, MobiGUITAR is an improvement upon AndroidRipper, utilizing the original concept of GUI ripping, but expanding it to allow users to explore even more different starting states by inputting different sets of values to be used during exploration [8, 12]. This time, compared to with AndroidRipper, MobiGUITAR was tested across four different applications, still using WordPress, but expanding to Tomdroid, Aard Dictionary , and Book Catalogue. The developers did a good

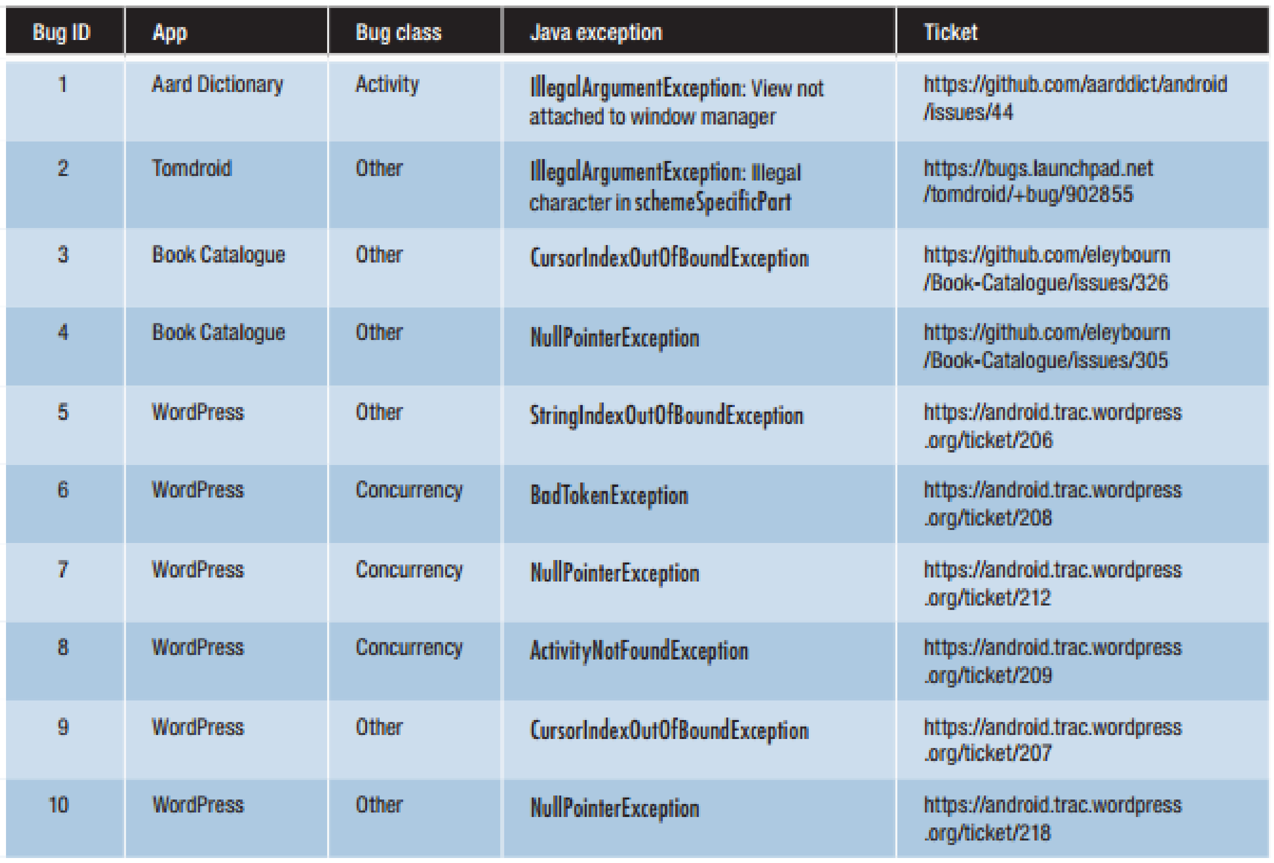
Table 6: Comparison of Results of Monkey and Dynodroid of the Unhandled Exceptions that MobiGUITAR Identified [12]



Bug ID	Found by	
	Monkey	Dynodroid
1	No	No
2	No	No
3	Yes	No
4	No	Yes
5	Yes	Yes
6	No	No
7	Yes	No
8	No	No
9	No	No
10	No	Yes

job of making sure to use the same version of Wordpress as they did on their previous tool making the comparison of results consistent. As a result, the application was and able to identify even more types beyond the original Activity and Event errors, such as Concurrency errors, as seen in Table 5 [1, 12]. In total,

Table 5: Bugs Identified by MobiGUITAR [12]



Bug ID	App	Bug class	Java exception	Ticket
1	Aard Dictionary	Activity	IllegalArgumentException: View not attached to window manager	https://github.com/aarddict/android/issues/44
2	Tomdroid	Other	IllegalArgumentException: Illegal character in schemeSpecificPart	https://bugs.launchpad.net/tomdroid/+bug/902855
3	Book Catalogue	Other	CursorIndexOutOfBoundException	https://github.com/eleybourn/Book-Catalogue/issues/326
4	Book Catalogue	Other	NullPointerException	https://github.com/eleybourn/Book-Catalogue/issues/305
5	WordPress	Other	StringIndexOutOfBoundException	https://android.trac.wordpress.org/ticket/206
6	WordPress	Concurrency	BadTokenException	https://android.trac.wordpress.org/ticket/208
7	WordPress	Concurrency	NullPointerException	https://android.trac.wordpress.org/ticket/212
8	WordPress	Concurrency	ActivityNotFoundException	https://android.trac.wordpress.org/ticket/209
9	WordPress	Other	CursorIndexOutOfBoundException	https://android.trac.wordpress.org/ticket/207
10	WordPress	Other	NullPointerException	https://android.trac.wordpress.org/ticket/218

MobiGUITAR generated and executed almost 8,000 tests and found ten bugs, three of which had been previously undocumented [12].

Also, in addition to improving upon AndroidRipper, the authors compared their new application again to Monkey and to Dynodroid mentioned above in section 4.3. These results are available in Table 6. Both of these tools were selected due to their random based and event based testing [12]. The inability of Dynodroid to let users configure event input proved to be a hinderance, as it failed to find bugs where MobiGUITAR succeeded [6, 12].

4.5 ACTEve

As presented above, ACTEve is an algorithm, developed by Anand et. al, that uses concolic testing to systematically generate event sequences from end-to-end across the application. This method is able to reduce the effect of the path explosion problem by checking conditions at runtime and pruning subsumption relations [13]. This algorithm was tested against five applications and the run time was analyzed in Figure 12. Anand et. al do well to address threats to validity such as the internal classes that may change the application's behavior; the authors validate manually that only irrelevant classes are ignored and that their algorithm does not ignore unimportant event sequences. As such, the authors can conclude that ACTEve is more efficient than naive concolic execution [13]. This method allows it to have an improved run time of the Collider tool discussed in section 3.2.1, due to its ability reduce path explosion [5, 13]. However, this application still has room for improvement in this particular area as path explosion may still prove challenging with subsumption patterns with many widgets or multiple event input types [13].

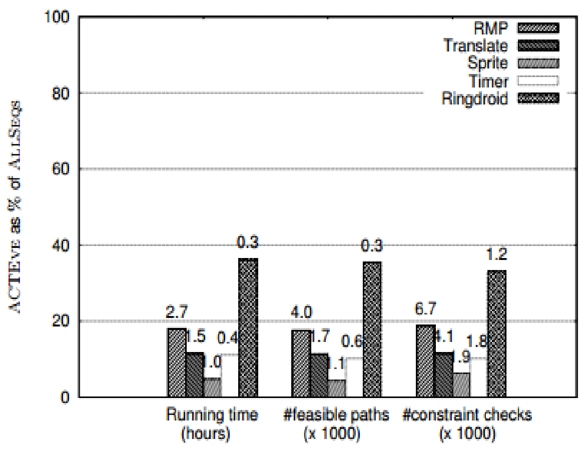


Figure 12: Results of ACTEve on Five Applications for Run Time [13]

In the third party analysis conducted by authors Choudhary et. al, it is noted that ACTEve is also able to generate system events much like Dynodroid [8].

5. Recommendations for Future Work

Overall, the field of automated testing for Android development still has a long way to go. An interesting avenue would be to combine the different testing techniques in single application to try to improve code coverage. For instance, random exploration based testing is able to identify more test cases quickly, and Adaptive Random Testing will help reduce redundancy in these test cases. If this technique was run in conjunction, perhaps prior to, concolic execution, which can target more complex test cases, the line coverage of an Android application could be substantially improved.

Additionally, some of these techniques have more room for improvement. One particular technique that was mentioned by several authors, but not often emphasized was fuzz testing. This is considered a subclassification of random testing and is actually implemented by Monkey. Fuzz testing strives to generate invalid inputs and to verify the robustness of the application rather than the functional correctness. There are several other fuzz testing tools and techniques out there that have improved upon Monkey, like a security testing framework capable of being run from the cloud that utilizes a tree structure similar to that of AndroidRipper [1, 15]. Another sample tool is IntentFuzzer which is able to detect how certain apps installed on a device will behave with other apps on the device, revealing potential security risks, by randomly generating null inputs [16].

Fuzz testing is actually considered effective for identifying security vulnerabilities, another classification of bugs not discussed by many of the authors of these papers [8]. Fuzz testing is limited in it's ability to generate specific inputs and thus may generate repetitive or redundant inputs and may not be able to traverse complex code paths. Perhaps for future work, fuzz testing code be integrated in some way, whether it is pre- or post-fuzzing, with concolic execution, to test high risk security applications. Anthony Wasserman has been one of the only authors to discuss the lack of progress with respect to security testing, despite the fact that mobile platforms are relatively open and vulnerable to new waves of malware [3]. Fuzzing is not completely worth overlooking as almost every single application compares their new tool or technique to Android's out of the box fuzzing tool, Monkey. In addition, if fuzzing should be considered a dated approach, then shouldn't Android have updated their standard SDK testing tool to use a different technique.

With respect to model based testing, the formation of the model, usually a state machine that is made from events and activities, is essential. AndroidRipper and MobiGUITAR were not tested as extensively as some of the other applications discussed in this paper; some authors found AndroidRipper to be under performing and unable to handle certain test cases. One suggestion for improvement is improving the reverse engineering process used to generate models. Several new techniques have already been proposed. Authors W. Yang, M. Prasad, and T. Xie have proposed ORBIT, a static analysis approach, as opposed to dynamic, that records set of events performed on the GUI and then is able to reverse engineer a model using crawling the app while the events are performed live [18]. This approach generates higher quality models with improved better code coverage in a shorter amount of time, having only generated relevant inputs [8]. This tool was specifically compared to AndroidRipper and found to about halve the runtime and double the code coverage [1, 18]. Model-based testing tools, specifically MobiGUITAR, since it is the evolved form of AndroidRipper, should look at the benefits and drawbacks of using static and dynamic analysis approaches to generating GUI models. This is just now being looked at with regards to Android applications as a whole, primarily with respect to security testing, not GUI testing [19].

Additionally, in their independent third party study of several of the testing tools discussed above, authors Choudhary et. al noted that there are no prevalent testing tools that have been sufficiently evaluated that can identify issues other than runtime exceptions [8]. The closest tool that is able to detect bugs using oracles, or a built-in library of user interactions and features supported by an application, is QUANTUM, developed by Zaeem et. al in 2014 [20]. This tool is able to use what it knows about an application based on the libraries it has been supplied with to predict and identify potential sources failure, much like a static analysis tool. This is another source of improvement where Android testing tools could strive to implement both static and dynamic analysis tools to increase code coverage.

Finally, a lot of these studies and testing tools have become dated as Android technology has continued to progress. With the new Android Runtime environment replacing Dalvik, the opportunity for automation of testing tools has expanded even more. Improved debugging tools packaged in with Android by default may lead to better results or solutions with minimal adaptations to the testing tools. One example of a more informative error message is a warning about the type of data attempting to be accessed when the program hits a null pointer exception [14]. Testing tools will be able to use this additional information to come up with more testing situations or to uncover even more bugs in the code.

Work continues to be done on testing tools in the Android development suite. As Android continues to mature, more tools will be required to help create efficient and clean programs with ease. As the platform continues to gain market share it becomes increasingly important to create an enticing environment for developers to use as they continue to make the next big app.

6. ACKNOWLEDGMENTS

Our thanks to Dr. Tim Menzies and George Matthews for a wonderful semester!

7. REFERENCES

[1] Using GUI Ripping for Automated Testing of Android Applications D. Amalfitano, A. Fasolino, S. Carmine, A. Memon, and P. Tramontana. Using GUI ripping for automated testing of Android applications. In Proceedings of 27th Intl. Conf. on Automated Software Engineering (ASE), 2012.

[2] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In Proceedings of the 6th International Workshop on Automation of Software Test (AST '11). ACM, New York, NY, USA, 77-83.

[3] A.Wasserman, Software Engineering Issues for Mobile Application Development, Proc. of the FSE/SDP workshop on Future of software engineering research, FOSER 2010, IEEE Comp. Soc. Press, pp. 397- 400

[4] Zhifang Liu, Xiaopeng Gao and Xiang Long. 2010. Adaptive Random Testing of Mobile Application. In Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET '10), IEEE Computer Society, Washington, DC, USA, 2, 297-301.

[5] Casper S Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, pages 67-77.

[6] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. Technical report, Program Analysis Group, Georgia Tech, 2012.

[7] Yang, S., Yan, D., and Routev, A. 2013. Testing for poor responsiveness in Android applications. In Proc. International Workshop on the Engineering of Mobile-Enabled Systems. MOBS '2013. 10-20.

[8] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? In Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015.

[9] Android Developers. The Developer's Guide.https://developer.android.com/guide/components/activities.html, last accessed on December 5th, 2016.

[10] T. Y. Chen, H. Leung, and I. K. Mak. "Adaptive random testing," In Advances in Computer Science: Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004), volume 3321 of M. J. Maher (ed.), Lecture Notes in Computer Science, pages 320–329. Springer, Berlin, Germany, 2004.

[11] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In ICSE, pages 396–405, 2007.

[12] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. In IEEE Software, vol. 32, pp. 53–59, 2015.

[13] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In Proc. 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2012.

[14] Android Developers. ART and Dalvik. https://source.android.com/devices/tech/dalvik/, last accessed on December 6th, 2016

[15] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of Android applications on the cloud. In Proceedings of 7th IEEE/ACM Workshop on Automation of Software Test (AST), 2012.

[16] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting Intents of Death," in Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), ser. WODA+PERTEA 2014. New York, NY, USA: ACM, 2014, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/2632168.2632169

[17] Android. Android - History. https://www.android.com/history/, last accessed on December 1st, 2016.

[18] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In FASE, 2013.

[19] Li Li, Tegawend´e F Bissyand´e, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein,

and Yves Le Traon. Static analysis of android apps: A systematic literature review. Technical report, 2016.

[20] R. N. Zaeem, M. R. Prasad, and S. Khurshid, “Automated generation of oracles for testing user-interaction features of mobile apps,” in Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, ser. ICST ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 183–192. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.31