Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android

Farnaz Behrang
Georgia Tech
Atlanta, GA, USA
behrang@gatech.edu

Alessandro Orso
Georgia Tech
Atlanta, GA, USA
orso@cc.gatech.edu

ABSTRACT

Experience paper: Testing of mobile apps is time-consuming and requires a great deal of manual effort. For this reason, industry and academic researchers have proposed a number of test input generation techniques for automating app testing. Although useful, these techniques have weaknesses and limitations that often prevent them from achieving high coverage. We believe that one of the reasons for these limitations is that tool developers tend to focus mainly on improving the strategy the techniques employ to explore app behavior, whereas limited effort has been put into investigating other ways to improve the performance of these techniques. To address this problem, and get a better understanding of the limitations of input-generation techniques for mobile apps, we conducted an in-depth study of the limitations of Monkey–arguably the most widely used tool for automated testing of Android apps. Specifically, in our study, we manually analyzed Monkey's performance on a benchmark of 64 apps to identify the common limitations that prevent the tool from achieving better coverage results. We then assessed the coverage improvement that Monkey could achieve if these limitations were eliminated. In our analysis of the results, we also discuss whether other existing test input generation tools suffer from these common limitations and provide insights on how they could address them.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Android UI testing, test generation, empirical study

ACM Reference Format:
Farnaz Behrang and Alessandro Orso. 2020. Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416567

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416567

1 INTRODUCTION

The use of mobile apps has become increasingly widespread. Due to the fierce competition in the mobile-app market, app quality is a primary concern and an important factor for the success of an app. Like for most software, testing is one of the crucial phases of the app development process, as it helps developers improve the quality of their app by revealing bugs before the apps are delivered to end users. Because users typically access the features provided by a mobile app through its graphical user interface (UI), app testing is also usually performed mainly through the UI of the app. Unfortunately, testing UI software is time-consuming and involves a great deal of manual effort. For this reason, researchers and practitioners alike have proposed a number of techniques and tools for automated test input generation for mobile apps (e.g. [1, 6, 14, 20, 23–25, 34, 38, 39, 41]).

Although effective, these techniques and tools are still limited in terms of the coverage they can achieve on the app under test. After studying these techniques, we believe that one overarching and common issue with them is that they tend to focus on improving their exploration strategy and overlook other important aspects of input generation. In fact, despite the large body of research on test input generation for mobile apps, little effort has been invested into a broader exploration of the limitations of (and improvement opportunities for) these techniques. We believe that a better understanding of these limitations can be extremely beneficial and provide meaningful insight on how to improve input generation for mobile apps. In this spirit, and as a first step in this direction, we performed an in-depth study of the performance of Monkey [5], a tool that automatically tests apps by generating pseudo-random streams of events. We chose Monkey as a representative technique because it is arguably the most widely used input-generation tool for Android apps and because previous studies [15, 36] have shown that it can outperform most other existing input generation tools in both open-source and industrial settings.

It is worth noting that, in related work, Zeng and colleagues [42] conducted an empirical study in which they studied Monkey's limitations by applying it to WeChat (https://www.wechat.com), a popular messenger app. Their study is limited, however, by the fact that it focuses on a single app. In this paper, conversely, we present the results of an extensive, in-depth study of Monkey's limitations. The goal of our study is twofold. *First*, we wanted to identify the inherent limitations of Monkey that prevent it from achieving better coverage results. In particular, our study focused on identifying limitations that are *common across many apps* and *go beyond limitations in the exploration strategies. Second*, we wanted to assess the coverage improvements that Monkey could achieve by eliminating these common limitations. Note that we focused on

coverage alone because (1) coverage is most commonly used as a proxy for the effectiveness of input generation techniques, and (2) we wanted to have a more focused analysis and room to present our results in greater detail. However, we plan to investigate other metrics, such as fault detection ability and possibly other kinds of coverage, in future studies.

In our study, we manually analyzed Monkey's performance on a benchmark of 64 apps from F-droid [18], a widely-used free and open-source Android app repository. (We initially considered 68 apps, but 4 of those no longer worked in modern versions of Android, so we discarded them.) We considered these apps because they have been used extensively in previous research work and have almost become a de-facto standard benchmark for evaluating the effectiveness of Android test input generation tools. Moreover, our analysis involved a considerable amount of manual inspection of the source code of the apps considered, and most industrial apps in the Google Play store are closed-source and obfuscated. It is also worth noting that the set of apps we studied contains some widely-used, large apps, such as WordPress, Wikipedia, K-9 Mail, and MyExpenses, which are available in the Google Play store and have been installed million of times. Whenever possible, we used the most recent version of the apps, as considering a newer version of an app allows us to get results that reflect recent features and enhancements in the Android operating system.

By applying Monkey to the 64 apps considered, we identified 7 categories of limitations that prevent Monkey from achieving better coverage results on 41 of these apps. The identified categories are *External apps*, *UI actions*, *Domain knowledge*, *Content providers*, *Input files*, *System events*, and *Inputs*, which prevent Monkey from achieving higher coverage for 34%, 34%, 22%, 15%, 15%, 10%, and 10% of the apps, respectively. Besides these general categories, we also found other limitations specific to individual apps. We consider these specific limitations less interesting, as addressing them is unlikely to provide general improvements.

After identifying and categorizing the above limitations, we wanted to investigate whether addressing them would actually benefit input generation. To do so, we semi-automatically eliminated these limitations by modifying the apps, as explained in Section 3, and assessed whether this improved the performance of Monkey on the 41 apps affected by the limitations. Doing so resulted in coverage improvements for 39 of the 41 apps ranging from 3% to 63%. More precisely, for these 39 apps, the average, median, and standard deviation of the coverage improvements were 25%, 24%, and 16%, respectively. For the remaining two apps, we were unable to assess the potential improvements, as it was not possible to modify them so as to eliminate the identified limitations.

In summary, our main findings are the following: (1) inter-app communication is common, especially across newly developed apps, and can considerably limit the effectiveness of input generation tools; (2) to explore the apps extensively, an input generation tool should go beyond performing simple random clicks, should be able to infer the most relevant UI actions, and should support performing other actions, such as long clicks, drag and drop, tap and hold, touch, swipe, multi-touch, and even voice commands; (3) domain knowledge is at times necessary to explore some behaviors of the apps under test, so input generation techniques must find ways to incorporate such domain knowledge; (4) populating content providers (or generating smart mocks) can also be necessary to test some behaviors of the apps; (5) input files often have a huge impact on the level of coverage that can be achieved on some apps (e.g., music or video player); (6) injecting system events, in addition to UI events, can considerably improve the effectiveness of input generation tools; and (7) specific inputs are at times required to explore particular behaviors of some apps, which implies that input generation tools should be able to generate context-aware inputs in addition to random ones.

Overall, we believe that our results support our intuition that there are important limitations for input-generation techniques that are orthogonal to the exploration strategies used. Developers of these kinds of techniques should be aware of these limitations and take them into account to improve the effectiveness of their techniques and tools. In our analysis of the results, we also provide insight on whether other existing test input generation techniques and tools consider these common limitations and, if they do not, how they could address them.

This paper makes the following contributions:

- The first extensive, in-depth study of the limitations of a representative and commonly used input-generation tool for mobile apps. Our results are available at [12].
- The identification of a set of limitations that are common across a majority of the apps considered.
- Empirical evidence that addressing these limitations can result in better coverage results.
- A detailed analysis of our empirical results that provides insights on whether other automated test input generation tools address the limitations we identified and, if not, how they could do so.

2 BACKGROUND

In this section, we present an overview of the Android platform, Android components, and Android test input generation tools.

2.1 Android Platform

Android apps are mainly written using the Kotlin and Java languages, but there are some platform libraries that allow developers to use C and C++ to achieve low latency or run computationally intensive apps, such as games or physics simulations. The Android SDK tools [4] compile source code along with any data and resource files into an Android Package Kit (*APK* for short). An *APK* file contains all the contents of an Android app and is used for the distribution and installation of the app.

At runtime, Java classes are converted into DEX bytecode, which is then translated to native machine code via ART [2] or Dalvik [3], two alternative runtimes. ART was introduced in Android 4.4 (KitKat) and has completely replaced Dalvik in Android 5.0 (Lollipop). Android 7.0 added to ART a JIT (just in time) compiler with code profiling capabilities to improve the performance of Android apps as they run.

2.2 Android Components

Components are the essential building blocks of an Android app and can be of one of four types: *Activities*, *Services*, *Broadcast receivers*, and *Content providers*.

Activities represent the user-facing screens of an app. Using activities, developers can place and organize UI components. Each activity provides callback methods that allow it to control the user interactions (e.g., clicks) with the screens.

Services allow apps to perform long-running background tasks or to perform work for remote processes without a user interface.

Broadcast receivers allow registering for system or app events. All registered receivers for an event are notified by the Android runtime once the event happens. For instance, apps can register for the system event that is fired once the device starts charging.

Content providers act as a central repository for storing app data and allowing other apps to access them. For instance, the Android system manages contacts through a content provider that can be queried by any app with the proper permissions.

2.3 Android Test Input Generation Tools

There is a large body of research on test input generation for Android apps, and researchers have proposed a number of test input generation tools for automating Android UI testing (e.g., [1, 6, 14, 20, 23–25, 30, 33–35, 38–41]). The goal of these tools is to explore as much behavior of apps as possible and reveal faulty behavior.

Android apps are event-driven programs so their inputs are typically in the form of UI events (e.g., clicks, swipe, and text inputs) or system events (e.g., receiving calls and text messages). Test input generation tools tend to generate such inputs using different exploration strategies, namely random, systematic, and model-based. Typically, *random* strategies generate purely random UI and system events. Conversely, *systematic* strategies use more sophisticated techniques (e.g., symbolic execution and evolutionary algorithms) to guide the exploration towards previously uncovered code. Finally, *model-based* strategies rely on a UI model of the app, often consisting of a finite state machine constructed statically or dynamically, to guide the exploration of the app and generate events.

3 EMPIRICAL STUDY

To study the limitations that prevent Android test input generation tools from achieving higher coverage, we performed an in-depth analysis of Monkey [5]. We chose Monkey because it is a widely used tool for testing Android apps, partly because it is part of the Android developers toolkit, which makes it is easy to use and compatible with different Android platforms and application settings [15]. In addition, studies conducted in both open-source and industrial settings confirm that Monkey performs well when compared with other existing test input generation tools [15, 36]. We used code coverage as our metric as it is commonly used by similar studies as a proxy for the effectiveness of input generation techniques, In future work, we plan to investigate additional metrics, such as fault detection ability.

Our empirical study aims to answer three research questions:

(1) **RQ1:** What are the general limitations, common across apps, that prevent Monkey from achieving higher coverage?
(2) **RQ2:** How much increase in coverage can Monkey achieve by eliminating these common limitations?
(3) **RQ3:** How can other automated test input generation tools address these common limitations?

3.1 Mobile App Benchmarks

To answer our research questions, we chose a benchmark of 68 apps from F-droid [18], a popular open-source Android app repository. These apps were initially collected by Choudhary and colleagues [15] to compare different Android test input generation tools. We chose these apps because they have been widely used in previous research (e.g., [7, 23–25, 28]) and have almost become a de-facto standard benchmark for evaluating the effectiveness of the Android test input generation tools. Moreover, our analysis involved a considerable amount of manual inspection of the source code of the apps considered, and most industrial apps in the Google Play store are closed-source and obfuscated.

Whenever possible, we used the most recent version of the apps, as considering a newer version of an app allows us to get results that reflect recent features and enhancements in the Android operating system (OS). Among the 68 apps that we studied, we were able to find a newer version for 19 apps (28%).

Table 1 lists the 68 apps that we considered in our study. For each app, the first four columns of the table show its name, category, version used in previous work, and new version used in our study. The cases for which a newer version of the app was not available are indicated with a dash ("-").

3.2 Experiment Setup

We conducted our study on a 64-bit Ubuntu 18.04 physical machine with 8 cores (3.40GHz Intel(R) CPU) and 32GB RAM. To run the apps, we used Android a set of x86 emulators, where each emulator was configured with 4 CPU cores, 2 GB of RAM, 1 GB of SD card, and the KitKat version of the Android OS (API level 19). If the app under study required a feature not available on the emulators, it was tested on a Samsung Galaxy S6 device running Android 7.0 (API level 24).

3.3 RQ1

To answer RQ1, we first ran Monkey for one hour on each benchmark app and collected line coverage using JaCoCo (Java Code Coverage Library [29]). We chose one hour because previous work [15] found that test input generation tools for Android apps typically achieve their maximum coverage within this time limit. Because many testing tools and apps are non-deterministic, we repeated each experiment 10 times and reported the maximum value across all runs. Note that we chose the maximum value, rather than the mean value, because our goal is not to compare Monkey with other existing tools, but rather to identify the limitations that prevent Monkey from achieving higher coverage. The resulting coverage information is shown in column "Coverage" in Table 1.

Once we collected coverage information for all benchmark apps, we manually analyzed each app along with its source code to assess why Monkey was unable to achieve higher coverage on that app. Column 6 in Table 1 lists the (common) Monkey limitations that we identified in our analysis.

Among the 68 apps that we analyzed, 4 no longer worked in newer versions of the Android OS, so we did not further consider them in the study: *Photostream*, *SyncMyPix*, *aagtl*, and *Mirrored*). For the remaining 64 apps, we were able to identify 7 categories of common limitations across 41 apps that prevented Monkey from

Table 1: List of the apps in our benchmark and of the limitations we identified that prevented Monkey to achieve higher coverage on these apps. A "" in the "New version" column indicates the apps for which a newer version was not available. In the "Limitations" column, "" indicates apps for which either no limitations were identified or the limitations were app-specific. A "*" in the same column indicates apps that no longer worked in modern versions of the Android OS and were therefore discarded.

Name	Category	Old version	New version	Coverage	Limitations
A2DP Volume	Transport	2.8.11	2.13.0.4	43.61%	System events (Bluetooth)
aagtl	Tools	1.0.31	-	32.73%	*
AardDictionary	Reference	1.4.1	2.43	40.83%	Input files (dictionary)
aCal	Productivity	1.6	-	14.83%	Inputs (configure server)
Addi	Tools	1.98	-	13.46%	Domain knowledge (Matlab commands)
ADSdroid	Reference	1.2	1.7.2	41.88%	-
aGrep	Tools	0.2.1	-	64.2%	Input files (text file), UI actions (long click)
Alarm Clock	Productivity	1.51	2.7	76.61%	UI actions (long click)
aLogCat	Tools	2.6.1	-	72.18%	-
Amazed	Casual	2.0.2	-	80.64%	System events (screen rotation)
AndroidomaticK.	Communication	1	-	97.74%	-
AnyCut	Productivity	0.5	-	69.97%	Content providers (contacts)
AnyMemo	Education	8.3.1	10.11.4	25.78%	External apps (Google Drive, Dropbox, Quizlet)
Auto Answer	Tools	1.5	-	33.5%	System events (phone call, Bluetooth)
BaterryDog	Tools	0.1.1	-	73.34%	-
Battery Circle	Tools	1.81	-	85.75%	-
Bites	Lifestyle	1.3	-	37.21%	System events (SMS), External apps (Trolly), UI actions (long click)
Blokish	Puzzle	2	3.2	48.54	Domain knowledge (game), UI actions (swipe, drag and drop)
Bomber	Casual	1	-	77.93%	Domain knowledge (game)
BookCatalogue	Tools	1.6	5.2.0	28.39%	External apps (LibraryThing, Amazon, Goodreads, Zxing, Pic2shop, sharing)
CountdownTimer	Tools	1.1.0	-	68.29%	Domain knowledge (letting alarm go off)
DalvikExplorer	Tools	3.4	-	68.8%	-
Dialer2	Productivity	2.9	-	41.12%	Content providers (contacts)
Divide&Conquer	Casual	1.4	-	82.94%	Domain knowledge (game)
FTP Server	Tools	2.2	3.0.1	17.4%	External apps (FTP client)
FileExplorer	Productivity	1	-	53.02%	-
FrozenBubble	Puzzle	1.12	-	73.33%	Domain knowledge (game), External apps (Google play store)
Gestures	Sample	1	-	54.83%	UI actions (touch)
HNDroid	News	0.2.1	-	35.65%	-
HotDeath	Card	1.0.7	-	72.48%	Domain knowledge (game)
ImportContacts	Tools	1.1	-	40.00%	Content providers (contacts), Input files (vCard file)
Jamendo	Music	1.0.6	-	68.1%	External apps (sharing), UI actions (long clicks)
K-9Mail	Communication	3.512	5.6	5.12%	Content providers (email accounts, email contacts), Inputs (login)
KeePassDroid	Tools	1.9.8	2.4.1	10.88%	UI actions (long click)
LearnMusicNotes	Puzzle	1.2	-	39.92%	Domain knowledge (game)
LockPatternGen	Tools	2	-	87.2%	-
LolcatBuilder	Entertainment	2	-	24.74%	Input files (images), External apps (sharing), UI actions (drag and drop)
Manpages	Tools	1.7	-	83.1%	-
Mileage	Finance	3.1.1	-	48.46%	UI actions (long click)
MiniNoteViewer	Productivity	0.4	-	77.24%	-
Mirrored	News	0.2.3	0.2.9	41.46%	*
Multi SMS	Communication	2.3	2.8	66.94%	-
MunchLife	Entertainment	1.4.2	-	88.66%	-
MyExpenses	Finance	1.6.0	3.6.5	45.65%	External apps (sharing, calendar, Dropbox)
myLock	Tools	42	-	36.18%	System events (phone call), UI actions (swipe)
Nectroid	Media	1.2.4	-	72.08%	-
NetCounter	Tools	0.14.1	-	72.67%	-
PasswordMakerPro	Tools	1.1.7	2.0.12	60.54%	-
Photostream	Media	1.1	-	8.5%	*
QuickSettings	Tools	1.9.9.3	-	52.19%	UI actions (tap and hold)
RandomMusicPlay	Music	1	-	90.76%	-
Ringdroid	Media	2.6	2.7.4	74.64%	Content providers (contacts)
Sanity	Communication	2.11	-	32.57%	External apps (Email, Google Play Store), system events (SMS, phone call, Bluetooth, Airplane mode)
SoundBoard	Sample	1	-	99.28%	-
SpriteMethodTest	Sample	1	-	84.82%	-
SpriteText	Sample	-	-	64.79%	-
SyncMyPix	Media	0.15	-	15.44%	*
TippyTipper	Finance	1.1.3	-	81.68%	-
TomdroidNotes	Social	2.0a	-	51.93%	Input files (Tomboy notes)
Translate	Productivity	3.8	-	76.92%	External apps (Email)
Triangle	Sample	-	-	90.09%	-
WeightChart	Health	1.0.4	-	74.76%	External apps (Email, Google Play Store), UI actions (long click)
WhoHasMyStuff	Productivity	1.0.7	1.0.18	78.77%	Content providers (contacts), External apps (calendar), UI actions (long click)
Wikipedia	Reference	1.2.1	2.7.5	56.75%	Inputs (login), External apps (sharing)
WordPress	Productivity	0.5.0	11.3	5.69%	Inputs (login), Input files (images, videos), External apps (social media)
World Clock	Tools	0.6	-	95.23%	UI actions (long click)
Yahtzee	Casual	1	-	55.04%	Domain knowledge (game)
ZooBorns	Entertainment	1.4.4	1.4.5	75.57%	-

achieving higher coverage. These categories are *External apps*, *UI actions*, *Domain knowledge*, *Content providers*, *Input files*, *System events*, and *Inputs*, which affected Monkey's performance on 34%, 34%, 22%, 15%, 15%, 10%, and 10% of the apps, respectively.

In addition to the categories that we identified, we also found other limitations that were more specific to individual apps. Consider, for instance, Tippy Tipper, which is a tip calculator app that allows for interacting with some UI elements using keyboard keys. For this app, using keyboard keys is required to cover much functionality of the app. As another example, consider MunchLife, an app for keeping track of character levels while playing a particular card game. To cover different scenarios, and thus achieve higher coverage for the app, Monkey needs to set the max level in the app preferences to different values. Because these kinds of limitations are very specific to the apps they affect, and we were looking for common issues that could benefit input generation tools in a more general way, we did not further consider them in our study. Similarly, we did not focus on other standard language issues that may affect coverage, such as dead code and special cases (e.g., exceptions and catch blocks).

The sixth column in Table 1 shows which apps were affected by which of the seven categories of limitations we identified. We marked the four apps we had to discard with a star ("*"), whereas a dash ("-") indicates those apps whose coverage was not affected by any of these seven categories. In the rest of this section, we discuss the seven identified categories in detail.

3.3.1 External apps. Communicating with external apps is needed to explore some functionality of 14 apps (34%) that we analyzed. Among these 14 apps, 5 needed access to at least a sharing app (e.g., messaging apps), 3 needed access to the Google Play Store, and 3 explicitly needed access to an email app. Some of the apps also needed to communicate with an FTP client app, Google Drive, Dropbox, Quizlet, LibraryThing, Amazon, Zxing or Pic2shop (barcode scanner apps), Goodreads, or Calendar.

Specifically, the set of apps that need access to sharing apps includes: LolcatBuilder, an app for sharing photos the user modifies; Jamendo, an app for sharing music; Wikipedia, an app for sharing Wikipedia articles; MyExpenses, an app for managing expenses and income, including sharing a PDF of transactions via email or Dropbox and spreading the word about the app through email; and BookCatalogue, an app for sharing books and sending information about the app to others through a sharing app.

FrozenBubble, Sanity, and WeightChart are the apps that need access to the Google Play Store to install other relevant apps or leave comments.

The apps that explicitly need access to an email app are Translate, Sanity, and WeightChart. Translate is a translation app that uses the Google Translation service. The app allows sharing the translations by email. Sanity provides the option to ask questions to the app developer by email. WeightChart is an app for keeping a personal log of body weight and displaying it as a chart. Also in this case, the app allows for contacting the app developer.

Apps that need to communicate with other external apps are FTP Server, Bites, AnyMemo, BookCatalogue, WordPress, and WhoHasMyStuff. FTP Server is an app that can read, write, and backup any folders in an Android device. This app needs to communicate with an FTP client in order for much of its functionality to be explored. Bites, a cookbook app, relies on Trolly, a shopping list app, for adding recipe ingredients to a shopping list. AnyMemo is a flashcard learning app that allows users to download and upload flashcards from/to Google Drive, Dropbox, and Quizlet, and needs the corresponding apps to do so. Users can synchronize the BookCatalogue app with their Goodreads account and also search for data related to books on Amazon, Goodreads, and LibraryThing (a social cataloging app for storing and sharing book catalogs and various types of book metadata). The app also supports using two barcode scanner apps (Zxing and Pic2shop) to scan books and automatically add them. The WordPress app can connect to social media services (i.e., Facebook, LinkedIn, Tumblr, Twitter) to automatically share new posts. The WhoHasMyStuff app also needs access to a calendar app to add events for expected returns of lent items.

3.3.2 UI actions. Among the apps we analyzed, we found 14 apps (34%) for which using UI actions not supported by Monkey was necessary to achieve higher coverage. Specifically, besides clicks, tests would need to perform long clicks, as well as drag-and-drop, tap-and-hold, touch, and swipe actions.

In the LolcatBuilder app, while modifying photos, captions can be dragged and dropped. QuickSettings is an app that provides quick access to various Android system settings, such as WiFi, GPS, brightness, and volume controls. In this app, it is necessary to tap and hold a setting icon to move it between visible and hidden lists or to reorder it. In the Bites app, long clicks are needed to modify recipes, ingredients, and methods. In the Alarm Clock app, long clicks are needed to change the time and modify alarm notifications. In World Clock, an app that shows time in different time zones at once, long clicks are needed to modify the time zones to be shown. In the Jamendo app, it is necessary to perform long clicks to modify playlists. Gestures, a simple app for demonstrating and practicing gestures, requires touch events. In the MyLock app, swipe actions are required to cover the lock and unlock functionality. In the aGrep app, modifying directories requires long clicks. For the Blokish app, both swipe and drag-and-drop actions are required to play the game. KeePassDroid, a password manager app for the Android platform, requires long clicks to modify entries in different kinds of lists. Mileage, an app for tracking vehicle mileage, requires long clicks to cover functionality related to vehicles and history. The WeightChart app requires long clicks to modify weight entries. Finally, the WhoHasMyStuff app also requires long clicks to modify information about lent items.

In general, a rich set of UI actions is needed to cover part of the functionality in the apps mentioned above. The degree to which this issue affects code coverage is different across apps, as we discuss in more detail in Section 3.4.

3.3.3 Domain knowledge. Specific domain knowledge is required to achieve higher coverage on nine apps (22%). Among these apps, seven are games that require an understanding of the game rules and dynamics: Divide&Conquer, a game where balls are bouncing and must be constrained into increasingly small areas; Yahtzee, a dice game; HotDeath, a card game; Bomber, an arcade game; Blokish, a board game; FrozenBubble, a bubble shooting game; and LearnMusicNotes, a simple game for music-reading training.

The other two apps that require domain knowledge to be suitably tested are Addi and CountdownTimer. Addi creates a mathematical computing environment similar to the one offered by Matlab and Octave, and requires inputs with a specific syntax to exercise any actual functionality. CountdownTimer is an alarm app that requires setting alarms and letting them go off to cover a considerable part of the functionality of the app.

3.3.4 Content providers. Content providers are necessary to cover the behavior of six of the apps considered (15%).

Five of these apps require access to phone contacts, which are provided by the Contacts Provider, a component that is part of the standard contacts app and that makes data about contacts available to other apps. These apps are the following: AnyCut, an app for creating shortcuts for contacts; Dialer2, an app for making and managing phone calls; ImportContacts, an app for importing contacts into a device; Ringdroid, an app for associating recordings or specific ringtones to different contacts; and WhoHasMyStuff, an app that helps to keep track of lent items and leverage contact information gathered from a user's address book.

The sixth app in this group is K-9Mail, an email client. To use the various features of the app, one needs a valid email account, with non-empty mailboxes, and contacts. A content provider within another email app (e.g., Gmail) can be leveraged to get access to this information.

3.3.5 Input files. Input files are needed to achieve higher coverage for six of the apps considered (15%). The required input files consist of images, dictionaries, text files, vCard files, Tomboy notes, and videos, as described below.

LolcatBuilder is an app for adding captions to, saving, and sharing photos. It is not possible to exercise the features of this app unless the app has access to image files. AardDictionary is a dictionary and offline Wikipedia reader, whose functionality can be explored only if the app has access to one or more dictionaries in a specific format (Slob–https://github.com/itkach/slob), which are not provided with the app. aGrep is a text search app similar to the Unix utility `grep`. To explore the part of the app functionality that searches for patterns within files, aGrep needs access to text files. ImportContacts, the app that we also mentioned in Section 3.3.4, needs access to vCard files from which to import contacts. TomdroidNotes is a Tomboy (a desktop note-taking app) client for Android. The app does not create Tomboy notes, but rather allows for modifying existing notes. Therefore, in order to explore the functionality of the app, Tomboy notes must be present. WordPress is a website-builder and a blog-maker app. Some parts of the app functionality related to the use of various media needs input files such as images and videos to be explored.

For these apps, not having the required input files can seriously limit the effectiveness of test-input generation tools. Moreover, in most cases, these input files must have a specific format (e.g., vCard, Tomboy) and/or specific content (e.g., a string matching a pattern) to trigger relevant behavior in the app.

3.3.6 System events. For six of the apps considered (15%), generating system events is necessary to achieve higher coverage. Although Monkey can generate pseudo-random streams of system events, it only generates limited types of these events (e.g., volume-control events). The system events required to suitably exercise these apps are events related to screen rotation, Bluetooth pairing, sending and receiving SMSs, and making and receiving phone calls.

Specifically, Amazed is a game in which screen rotations trigger specific behaviors. A2DP Volume is an app for (1) managing the volume of an audio stream when the device on which the app is running is connected to a Bluetooth device and (2) restoring the volume on reconnect. Bites, a cookbook app we already described, allows users to create recipes and share them via SMS. AutoAnswer is an app that allows for answering the phone automatically when it rings. myLock is an app that enables quick unlock, in-call touch-screen lock, and incoming call prompts. Sanity is a phone assistant app with features such as caller announcement and call blocking.

3.3.7 Inputs. For four of the apps considered (10%) specific inputs must be provided to the apps to exercise parts, when not most of their functionality.

aCal is an app for accessing a calendar on a CalDAV server and managing tasks and events. Much of the functionality of this app can only be explored if the app has access to a calendar on a CalDAV server, which requires the user to have access to the server and to provide the app with a username, the corresponding password, and the URL of the server. Both the Wikipedia and the WordPress apps require users to log into the apps. In the Wikipedia app, logging into the app is only needed for accessing some functionality, while in the WordPress app it is required for exploring a large portion of the functionality. Finally, in the K-9Mail app, the user can also directly provide login information for an email server instead of accessing email accounts from other email apps through content providers.

Summary of RQ1. *What are the general limitations, common across apps, that prevent Monkey from achieving higher coverage?* Among the 64 apps that we analyzed, we were able to identify 7 categories of common limitations that affect the ability of the input-generation tool to achieve high coverage. These limitations affect Monkey's performance on 41 of the 64 apps overall, and between 10% and 34% of the apps individually. The identified categories are listed in Table 1, along with more details about the apps involved and their versions.

3.4 RQ2

To answer RQ2, we attempted to eliminate the common limitations we identified earlier (to the extent possible) and then collected the coverage after eliminating these common limitations.

For apps that required interactions with external apps, we set up those apps before running Monkey. For apps that required specific UI actions to explore part of their functionality, we guided Monkey to perform those actions on suitable UI elements. If an app required domain knowledge to cover some specific functionality, we provided the app with suitable human-created inputs. Specifically, for the game apps, we helped Monkey to reach subsequent levels in the games; we provided the Addi app, which reads commands in Matlab format, with commands provided as examples in the source code of the app; and we simply let alarms go off in the CountdownTimer app. For those apps requiring content providers, we suitably populated the required content providers before starting

Monkey. For those apps that needed access to particular input files, we made different input files available on the device before starting Monkey For the apps that required system events to perform part of their functionality, we injected the relevant events while Monkey was testing the app. Finally, for the apps that needed specific inputs, we either provided login information or suitably configured servers providing data to the app before running Monkey.

Figure 1 shows the initial and the improved coverage achieved by Monkey after addressing the common limitations listed in Table 1 in the way we just described. As the figure shows, coverage increased for 39 of the 41 apps affected by these limitations, with improvements ranging from 3% to 64%. For these 39 apps, the average, median, and standard deviation of the coverage improvement are 25%, 24%, and 16%, respectively. For the other two apps, AnyMemo and Translate, we were unable to achieve improvements, despite the fact that we addressed the common limitation relevant for these apps (i.e., accessing external apps). Specifically, the open-source version of AnyMemo was unable to connect to the provided external apps, whereas Translate crashed when trying to open an external email app.

We examined in more detail the apps for which the changes we made in how Monkey is run led to considerable coverage improvements. Monkey achieved over 60% increase in coverage for LolcatBuilder after (1) making images available on the device, (2) allowing the app to connect to a sharing app, and (3) supporting drag-and-drop actions. For two apps, K-9Mail, and LearnMusicNotes, the coverage increased between 50% and 60% by simply providing login information and domain knowledge. Monkey's coverage increased between 40% and 50% for three other apps (Dialer2, ImportContacts, and WordPress) by populating content providers, providing vCard, images, and video files, along with setting up social media apps and providing login information.

For seven of the apps (AardDictionary, FTP Server, Bites, Auto Answer, Yahtzee, Gestures, and KeePassDroid), coverage increased between 30% and 40% by (1) providing dictionary files, (2) injecting SMS-, phone-call-, and Bluetooth-related system events, (3) providing access to an FTP client and the Google Play Store, (4) supporting long clicks and touch events, and (5) injecting domain knowledge. It is worth noting that, although the Bites app needs access to a shopping list app called Trolly to keep track of recipe ingredients, this app does not seem to exist any longer, so we were not able to cover this functionality in the app.

AnyCut, aGrep, BookCatalogue, Mileage, and Sanity are among the apps for which the coverage increased between 20% and 30%. For these apps, the coverage increase is due to (1) the injection of SMS-, phone-call-, and Bluetooth-related system events, (2) enabling/disabling of airplane mode, (3) populating contacts, (3) providing text input files, (4) supporting long clicks, and (5) setting up the Google Play Store, LibraryThing, Amazon, Goodreads, Zxing, Pic2shop, sharing, and email apps.

For ten of the apps (Divide&Conquer, A2DP Volume, Jamendo, myLock, Bomber, Wikipedia, CountdownTimer, Ringdroid, TomdroidNotes, and WeightChart), coverage improved between 10% and 20%. In this case, what helped Monkey's input generation was (1) the use of domain knowledge, (2) injecting Bluetooth- and phone-call-related system events, (3) supporting long-click and swipe actions, (4) providing login information and Tomboy note files, (5) populating contacts, and (6) setting up the Google Play Store, email, and other sharing apps.

Finally, Amazed, QuickSettings, Alarm Clock, World Clock, HotDeath, FrozenBubble, MyExpenses, and WhoHasMyStuff witnessed coverage increases between 3% and 10% when (1) injecting screen-rotation system events, (2) adding long-click actions, (3) using domain knowledge, (4) populating contacts, and (5) setting up the Google Play Store, Dropbox, calendar, and other sharing apps.

Summary of RQ2. *How much increase in coverage can Monkey achieve by eliminating these common limitations?* By eliminating the common limitations we identified in our analysis, which are relevant for 41 of the 64 apps considered, we were able to increase the coverage achieved by Monkey for 39 of these apps. The coverage improvements range from 3% to 64%, with average, median, and standard deviation of 25%, 24%, and 16%, respectively. These results clearly indicate that input-generation tools can achieve considerable improvements in terms of coverage achieved and justify the investigation of techniques that can automatically address the identified limitations.

3.5 RQ3

Automated test input generation techniques and tools for Android apps are often focused on improving the UI exploration strategy. For this research question, we aim to identify other possible improvements that are worth pursuing based on our results for RQ1 and RQ2. In the rest of this section, we discuss whether (and how) can automated input generation tools other than Monkey address the common limitations we identified in Section 3.3.

3.5.1 External apps. As shown in our study, apps rely extensively on other apps (e.g., sharing apps) to implement their functionality. Furthermore, some of the apps we considered in our study have been developed a while ago; we believe that inter-app communications are becoming even more common among newly-developed apps. Android apps interact with one another through intents. Whereas existing testing frameworks allow for *manually* testing this type of interactions (e.g., by supporting the specification of mocks), to the best of our knowledge they provide no automated support for generating tests containing inter-app interactions. To be able to test interactions between apps, existing tools would have to be extended with the ability to automatically create and consume intents coming from and going to the app under test.

3.5.2 UI actions. When interacting with UI elements, it is important to identify which actions each UI element can receive and process. As our study shows, random test input generation tools, such as Monkey, might not necessarily perform the UI actions that are required to interact with a particular UI element. Other, more sophisticated test input generation tools use both static and dynamic analysis techniques to infer relevant UI actions in a given state or on a given screen. Dynamic analysis techniques typically infer actions by examining the UI hierarchy, whereas static analysis techniques tend to rely on the analysis of the event listeners in the source code of the app. Recently, approaches based on machine learning (e.g., QLearning-Based Exploration (QBE) [21] and reinforcement learning [16]) have been shown to be effective in identifying relevant UI actions.

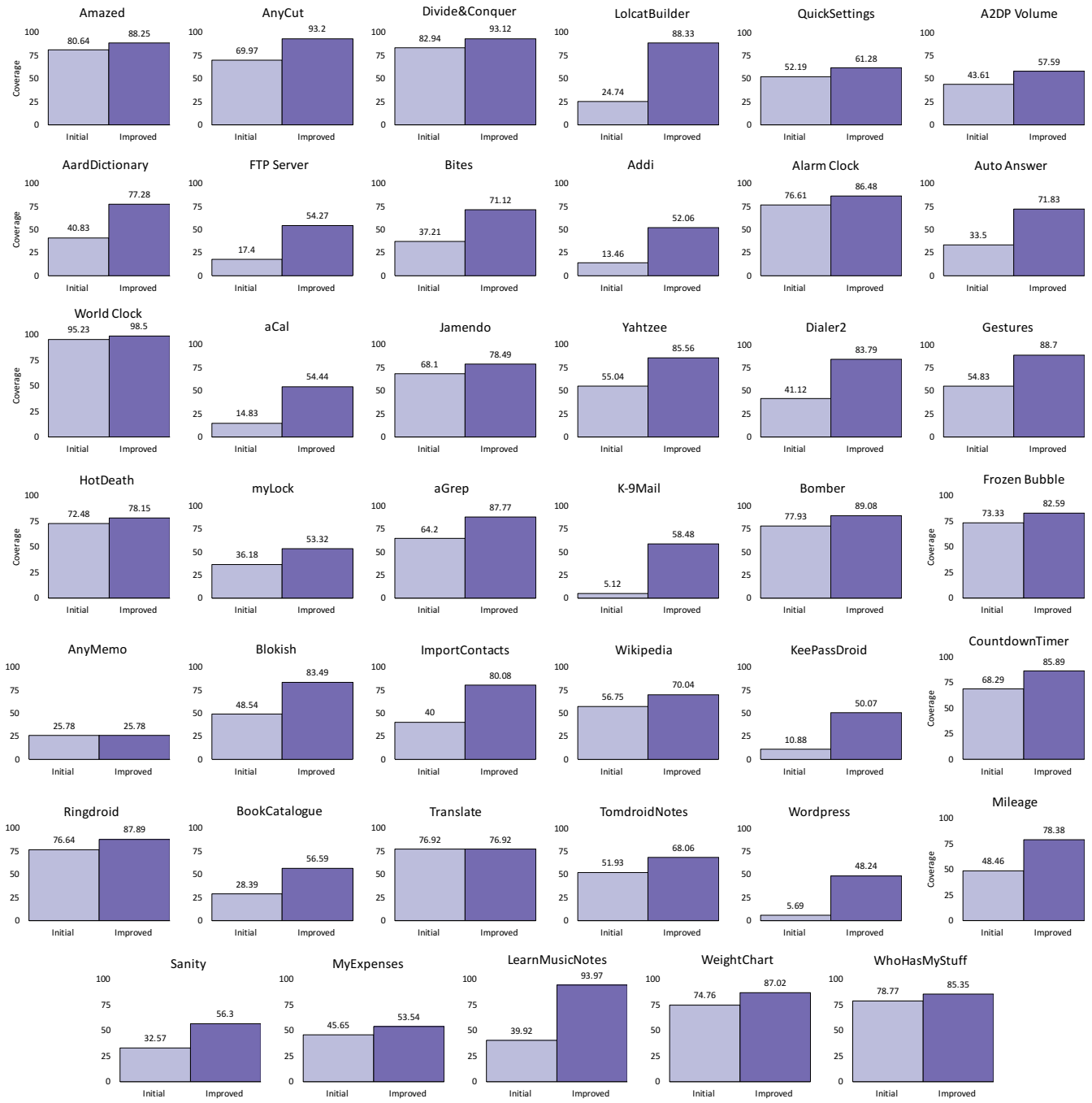

Figure 1: Bar charts showing, for the 41 apps affected by the common limitations listed in Table 1, the coverage achieved by vanilla Monkey and the improved coverage achieved by Monkey after we manually addressed these limitations.

In general, test input generation tools must go beyond randomly triggering actions to improve their effectiveness. Furthermore, in addition to basic actions, such as click, scroll, and swipe, test input generation tools should also support multimodal actions, such as multi-touch actions and voice commands; some apps, for instance, require users to draw a particular pattern on the screen to activate some functionality.

3.5.3 Domain knowledge. Some apps have behaviors that are extremely unlikely to be explored randomly, and testing those requires some form of domain knowledge. In this context, one particular class of apps are games. For testing game apps, also test input generation techniques other than random, such as search-based, coverage-guided, and symbolic-execution-based techniques, are also not particularly effective and fail to explore deep behavior in

the apps. Recently, there have been promising results in the use of deep reinforcement learning approaches for game testing [43]. However, these results are still preliminary, and more research in this direction is needed.

For apps other than games, domain knowledge related to the app or even its category could be leveraged to improve the effectiveness of testing tools. For instance, the source code of Addi, which requires Matlab commands as inputs, contains examples of such commands (as comments) that can be used to test the app. For another example, in the case of Ringdroid, allowing alarms to go off is not specific to this app and is true for all the apps that deal with alarms or timers. In fact, some recent work, including our own, have started to explore ways to leverage the human knowledge encoded into existing tests or provided by the crowd to improve UI testing (e.g., [11, 26, 32]). However, also in this case, existing research is still preliminary and more work needs to be done in this area.

3.5.4 Content providers. Our study shows that it is necessary to have suitably populated content providers in order to cover specific behaviors of some apps. Similar to the case of external apps, mock objects can be used to simulate the presence of content providers and test the apps that rely on them. In order for test input generation tools to effectively test these apps, therefore, they should be able to automatically mock relevant content providers (e.g., to simulate contacts or gallery images on a device or an emulator without actually adding them). To the best of our knowledge, however, none of the existing test input generation tools for Android support automatically mocking content providers.

3.5.5 Input files. It is very common for some app categories, such as music or video players, to require the presence of input files in order to be suitably tested. Although it would be theoretically possible to store a standard set of files (e.g., JPG, MP3, and PDF files) on a device, so that an app under test could access them if needed, some apps require very specific types of files. AardDictionary and TomdroidNotes, for instance, need access to particular dictionary files and Tomboy note files, respectively. It would not be possible to know which files these apps need without consulting their documentation. One possible way to address this issue would be to provide a way for developers to specify a minimum testing environment for their apps. Alternatively, test input generation tools could be extended to automatically generate mocks that can intercept calls to the filesystem, but this type of solution is unlikely to work in the presence of files with structured and complex content.

3.5.6 System events. For some apps, such as those mainly used for communication, system events drive a large fraction of the app behavior. For some other apps, system events play a less central, yet still important role. Only a few existing test input generation tools for Android have the ability to inject system events in addition to UI events (e.g., [1, 23, 34]). These tools analyze the apps to find system events the apps handle and generate specific Broadcast Intents to simulate these events. As the evaluation of Stoat [34] shows, injecting this type of events can considerably increase the number of crashes detected by an input generation tool.

3.5.7 Inputs. As our study shows, in some cases, specific inputs must be provided to an app to explore parts of its behavior. To test a browser app, for instance, valid website addresses should be used. For another example, the names of actual movies should be used to test a movie-search app. Some existing test input generation tools (e.g., [23, 39]) allow users to manually provide inputs that can help testing, such as login and password for an app that requires authentication. Some more recent techniques try to go beyond these approaches and automatically generate context-aware inputs [22], infer grammars for generating inputs with the right format [8, 19, 27], and allow users to specify simple input generators that can help fuzzing [31]. In general testing tools should try to go beyond generating purely random inputs and also allow users, as a fallback solution, to specify specific inputs when needed.

Summary of RQ3. *How can other automated test input generation tools address these common limitations?* Although most existing input generation tools either fail to address the limitations we identified in our study, there are ways in which at least some of these limitations can be addressed. Moreover, there are recent tools that, although still immature, aim to mitigate these limitations and represent important steps in the right direction.

4 THREATS TO VALIDITY

There are several threats to the *external validity* of our results. First, our results may not generalize to other apps. To mitigate this threat, we selected benchmarks that were used in previous research and that cover a broad range of app categories Second, for our experiments we used Android 4.4 KitKat (API level 19), which is not a recent version of the Android OS, because that allowed us to compile and run most apps. More empirical studies with additional benchmark apps and newer versions of Android would increase confidence in the external validity of the results. Regarding *internal validity*, we only measured coverage of the Java code within the apps, while ignoring parts written in other languages. Although Android apps are written mainly, when not exclusively, in Java, this may affect our results. Finally, regarding *construct validity*, we could have used different metrics to investigate our research questions. However, we believe that the ones we chose are appropriate for the questions that we were investigating and plan to consider other metrics (e.g., fault detection) in future work.

5 RELATED WORK

There has been a large body of research on automated test input generation for Android apps. In this section, we describe the work most closely related to ours, grouped by topic.

5.1 Empirical Studies

Choudhary, Gorla, and Orso [15] conducted an empirical study on publicly available tools that can automatically generate test inputs for Android apps. Their study investigated these tools and their underlying techniques to understand how they compare to one another and which ones may be more suitable in which context. Their study also focused on understanding the ways that the existing tools can be improved or new tools can be developed.

Wang and colleagues [36] conducted an empirical study in which they compared existing state-of-the-art Android test input generation tools, when used on industrial apps, in terms of both code coverage and fault detection. Their study reports their experience in applying these tools in an industrial setting and provides insights

on the strengths and weaknesses of the tools considered and on how combining some of these tools could improve code coverage or fault detection ability.

Our empirical study is different from this previous work, as our goal is not to compare the existing input generation tools, but rather to identify (1) common limitations of random test input generation, using Monkey as a reference approach, and (2) the effect of these limitations in terms of test coverage achieved.

Zeng and colleagues [42] conducted a study in which they applied Monkey to WeChat, a feature-rich messaging app, identified Monkey's limitations in this context, and reported their findings. They also developed an approach to address these limitations and achieved consequent improvements in terms of coverage. Our study is different because it focuses on identifying limitations of Monkey (and similar tools) that are not specific to one app, but rather common across apps and go beyond the limitations that are only related to exploration strategies.

5.2 Techniques and Tools

Test input generation tools can be classified into four categories based on their exploration strategies: random, model-based, systematic, and reinforcement-learning-based.

5.2.1 Random exploration strategy. These are test input generation tools that employ a random strategy to generate inputs for Android apps. Monkey [5] is the most commonly used of these tools. It is developed and supported by Google and is part of the Android Software Development Kit (SDK). Monkey generates pseudo-random streams of user events to test an app. Dynodroid [23] applies random testing, but it takes into account the context during exploration. It also injects system events that are relevant to the app by analyzing its source code. DroidMate [30] also uses a random strategy to explore apps without requiring root access or modifying the OS. Intent fuzzer [33] tests how an app can interact with other apps by statically analyzing the source code of the app, identifying which intents the app uses, and generating intents accordingly.

5.2.2 Model-based exploration strategy. Model-based strategies construct and use a model of the app under test to systematically generate inputs for the app. These models are often finite state machines in which nodes represent screens (i.e., activities and menus) and edges represent transitions between them. DroidBot [41] dynamically generates a state transition graph and uses the generated graph to guide the exploration of the app under test. GUIRipper [39] also builds a model of the app dynamically, by crawling it, and subsequently uses that model to explore the app. ORBIT [38] first statically analyzes the app to extract relevant UI events, then uses that information to build a model of the app, and then uses the model to test the app. A^3E-Depth-First and A^3E-Target [6] both build a model of the app and then use it to explore the app systematically. The former builds the model using static dataflow analysis, whereas the latter uses dynamic tainting. SwiftHand [14] creates a dynamic finite state machine model of the app and refines it to minimize the restarts of the app while exploring it. PUMA [20] is a generic framework that allows for implementing dynamic analyses of Android apps. Stoat [34] combines static and dynamic analysis to construct a model of the app and then, like the other techniques in this group, it uses the generated model to guide the exploration of the app during testing. Unlike other techniques, Stoat injects, in addition to UI events, system events, so as to improve the effectiveness of test input generation.

5.2.3 Systematic exploration strategy. Systematic exploration is a class of exploration strategies that use more sophisticated techniques, such as symbolic execution and evolutionary algorithms, to guide the exploration towards previously uncovered code. EvoDroid [24] uses an evolutionary algorithm whose fitness function aims to maximize coverage. ACTEve [1] aims to explore as many paths as possible using dynamic symbolic execution. IntelliDroid [37] is a generic Android test input generator that can be configured to produce inputs that target methods that are relevant for a specific dynamic analysis tool. CuriousDroid [13] decomposes the UI of the app under test on-the-fly and creates a context-based model for interactions that are tailored to the current screen layout. Sapienz [25] uses a multi-objective search-based approach to automatically explore and optimize test sequences, minimizing length, while simultaneously maximizing coverage and fault detection.

5.2.4 Reinforcement-learning-based exploration strategy. Koroglu and colleagues [21] use a well-known reinforcement learning technique called Q-Learning to explore UI actions within Android apps. Esparcia and colleagues [17] also use Q-learning as a metaheuristic for selecting UI actions in their automatic testing tool. Bauersfeld and Vos [9, 10] propose an approach for fully automating robustness testing of complex GUI applications that is based on Q-Learning and aims to combine the advantages of random and coverage-based testing. Degott, Borges, and Zeller [16] combine a crowd-based model with a reinforcement learning exploration strategy to automatically learn which interactions can be used for which UI elements. They then use this information to guide the test generation.

6 CONCLUSION AND FUTURE WORK

We presented an in-depth empirical study of the limitations of Monkey, a representative and widely used input generation tool for Android apps. In the study, we analyzed Monkey's performance on 64 apps and identified 7 general categories of limitations that can prevent Monkey from achieving better coverage results on 41 of the considered apps. We then manually eliminated the identified limitations and reapplied Monkey to these apps, which resulted in coverage improvements between 3% and 63% for 39 of the 41 apps. In our analysis of the results, we also discussed whether other existing Android test input generation techniques can suffer from the same limitations we identified for Monkey and, if so, provided insights on how these techniques could be improved. In future work, we will extend our empirical study by including additional benchmarks, and in particular industrial apps, so as to confirm and possibly refine our current results. In future studies, we will also consider additional metrics, such as fault detection ability and other kinds of coverage, and assess how that affects our results.

ACKNOWLEDGEMENTS

This work was partially supported by NSF, under grant CCF-1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, and gifts from Facebook, Google, and Microsoft Research.

REFERENCES

[1] Saswat Anand, Mayur Naik, Mary Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012*. ACM, New York, NY, USA.

[2] Android Open Source Project. 2020. Android Runtime (ART) and Dalvik. https://source.android.com/devices/tech/dalvik.

[3] Android Open Source Project. 2020. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[4] Android Open Source Project. 2020. SDK Platform Tools release notes. https://developer.android.com/studio/releases/platform-tools.

[5] Android Open Source Project. 2020. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.

[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 641–660.

[7] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 238–249.

[8] Bastani, Osbert and Sharma, Rahul and Aiken, Alex and Liang, Percy. 2017. Synthesizing Program Input Grammars. *SIGPLAN Not.* 52, 6 (June 2017), 95–110.

[9] Sebastian Bauersfeld and Tanja Vos. 2012. A Reinforcement Learning Approach to Automated GUI Robustness Testing. In *Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 7–12.

[10] Sebastian Bauersfeld and Tanja EJ Vos. 2014. User interface level testing with TESTAR; what about more sophisticated action specification and selection?. In *SATToSE*. CEUR-WS.org, Aachen, 60–78.

[11] Farnaz Behrang and Alessandro Orso. 2018. Automated Test Migration for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 384–385.

[12] Behrang, Farnaz and Orso, Alessandro. 2020. Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android. https://sites.google.com/view/studymonkeylimitations/.

[13] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2017. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Financial Cryptography and Data Security*, Jens Grossklags and Bart Preneel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 231–249.

[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 623–640.

[15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, New York, NY, USA, 429–440.

[16] Christian Degott, Nataniel P. Borges Jr., and Andreas Zeller. 2019. Learning User Interface Element Interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 296–306.

[17] Anna I Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda, and T Vos. 2016. Q-learning strategies for action selection in the TESTAR automated testing tool. *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)* (2016), 130–137.

[18] F-droid Group. 2020. F-Droid. https://f-droid.org.

[19] Godefroid, Patrice and Peleg, Hila and Singh, Rishabh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, New York, NY, USA, 50–59.

[20] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. ACM, New York, NY, USA, 204–217.

[21] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez. 2018. QBE: QLearning-Based Exploration of Android Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, New York, NY, USA, 105–115.

[22] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic Text Input Generation for Mobile Testing. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, New York, NY, USA, 643–653.

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234.

[24] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 599–609.

[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 94–105.

[26] K. Mao, M. Harman, and Y. Jia. 2017. Crowd Intelligence Enhances Automated Mobile Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. ACM, New York, NY, USA, 16–26.

[27] Mathis, Björn and Gopinath, Rahul and Mera, Michaël and Kampmann, Alexander and Höschele, Matthias and Zeller, Andreas. 2019. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 548–560.

[28] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 559–570.

[29] Mountainminds GmbH & Co. 2020. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/.

[30] N. P. Borges, M. Gómez, and A. Zeller. 2018. Guiding App Testing with Mined Interaction Models. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. ACM, New York, NY, USA, 133–143.

[31] Padhye, Rohan and Lemieux, Caroline and Sen, Koushik and Papadakis, Mike and Le Traon, Yves. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[32] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Efficient GUI Test Generation by Learning from Tests of Other Apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 370–371.

[33] Raimondas Sasnauskas and John Regehr. 2014. 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)* (San Jose, CA, USA) *(WODA+PERTEA 2014)*. ACM, New York, NY, USA, 1–5.

[34] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.

[35] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5.

[36] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 738–748.

[37] Michelle Wong and David Lie. 2016. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *NDSS16*. The Internet Society, Reston, VA, USA, 21–24.

[38] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering* (Rome, Italy) *(FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 250–265.

[39] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering* (Rome, Italy) *(FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 250–265. https://doi.org/10.1007/978-3-642-37057-1_19

[40] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia* (Vienna, Austria) *(MoMM '13)*. ACM, New York, NY, USA, 68–74.

[41] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-Guided Test Input Generator for Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE Press, New York, NY, USA, 23–26.

[42] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There yet in an Industrial Case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 987–992.

[43] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. 2019. Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 772–784.