Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?

Xia Zeng[1] Dengfeng Li[2] Wujie Zheng[1] Fan Xia[1] Yuetang Deng[1] Wing Lam[2] Wei Yang[2] Tao Xie[2]
[1]Tencent, Inc., China
[2]University of Illinois at Urbana-Champaign, USA
[1]{xiazeng,wujiezheng,frankxia,yuetangdeng}@tencent.com, [2]{dli46,winglam2,weiyang3,taoxie}@illinois.edu

ABSTRACT

Given the ever increasing number of research tools to automatically generate inputs to test Android applications (or simply apps), researchers recently asked the question "Are we there yet?" (in terms of the practicality of the tools). By conducting an empirical study of the various tools, the researchers found that Monkey (the most widely used tool of this category in industrial settings) outperformed all of the research tools in the study. In this paper, we present two significant extensions of that study. First, we conduct the first industrial case study of applying Monkey against WeChat, a popular messenger app with over 762 million monthly active users, and report the empirical findings on Monkey's limitations in an industrial setting. Second, we develop a new approach to address major limitations of Monkey and accomplish substantial code-coverage improvements over Monkey. We conclude the paper with empirical insights for future enhancements to both Monkey and our approach.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Android, GUI testing, test generation, code coverage

1. INTRODUCTION

Given the ever increasing abundance of tools [7, 3, 4, 5, 8, 10] to automatically generate inputs to test Android apps, especially from the research community, recently Choudhary et al. [6] asked the question "Are we there yet?" in terms of having good-enough tools for the concept to be used in practice. They conducted an empirical study on publicly available tools that can automatically generate inputs to test Android apps. The study was intended to assess these tools (and their underlying techniques) to investigate which tools may be better suited under which contexts (*e.g.,* app types), to understand how existing tools can be improved and what new tools should be developed. In addition to six research tools from the academia, the study also considered the open source tool from Google, Monkey[1], which is the most widely used tool in industrial settings, due to its applicability to a variety of application settings, *e.g.,* ease of use and compatibility with different Android platforms.

Despite forming valuable contributions as a starting point for the community, the study conducted by Choudhary et al. [6] can be further extended in two important ways. First, their study considered relatively simplistic, open-source apps as the apps under test (partly because some of the tools under comparison require the source code of the apps under test). No industrial-strength Android app was included in the study. Second, although the study reported that Monkey (achieving less than 50% code coverage) outperformed all five research tools, they did not empirically demonstrate whether and how new (hypothesized) techniques can further improve the code coverage achieved by Monkey.

Accomplishing these two extensions is challenging. For the first extension, there are many difficulties to empirically study tools such as Monkey on popular industrial-strength Android apps (*e.g.,* the Facebook app). For example, most (if not all) popular Android apps are close-source apps, but there is no robust code-coverage measurement tool on instrumenting Android apps without requiring access to the apps' source code. A natural solution to address this issue is to collaborate with the vendors of these industrial-strength Android apps (who have access to the source code) to apply a code-coverage measurement tool such as Emma [2]. However, based on our experiences, applying Emma on an industrial-strength Android app can cause Emma to trigger a 64K Reference Limit exception [1] during instrumentation. As for the second extension, based on our empirical observations of the limitations of existing tools, one can hypothesize various techniques to address these limitations. However, the high complexity of industrial-strength apps often requires a careful design or configuration of these techniques in order to empirically demonstrate their benefits.

To accomplish the first extension, we first study the effectiveness and limitations of Monkey to test WeChat[2], a highly popular messenger app (especially among users of Chinese origins) released by Tencent, Inc. WeChat is one of the most popular messenger apps in the world with over **762 million** monthly active users[3]. In fact, WeChat has

[1]https://developer.android.com/studio/test/monkey.html
[2]https://www.wechat.com
[3]http://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

evolved to be well beyond a messenger app: it also supports many functionalities such as banking, shopping, and serves as a platform for third parties to develop their own apps[4].

Our results from applying Monkey on WeChat reveal major limitations of Monkey's random exploration strategy. Due to Monkey's random nature, we initially expected that Monkey would not be able to achieve high coverage of code lines (involving likely complex logic) deep within an activity. Indeed, our study results confirm such expectation. Monkey covers only 19.5% of the code lines in WeChat. At the same time, we expected that Monkey would be able to achieve a much higher activity coverage than line coverage (an activity corresponds to a UI screen and reflects a coarse-grained app feature). The reason for such expectation is that we can make a rough analogy between activity coverage and class coverage, and class coverage is generally much higher than line coverage (covering a class is typically much easier than covering a particular line in a class). However, Monkey actually achieves surprisingly low activity coverage (10.3%). Furthermore, we observe that the majority of the events generated by Monkey are redundant events (*e.g.,* events repeatedly exercising no code behavior or exercising previously-exercised code behaviors). The main reasons for such observation are two-fold: (1) widget obliviousness: Monkey is oblivious to the locations of widgets on a screen; and (2) state obliviousness: Monkey is oblivious to the GUI states before or after an event, and thus cannot distinguish a state-changing event from a state-preserving event.

To accomplish the second extension, we develop a new approach that inherits the high applicability of Monkey while addressing its empirically-observed limitations. Our empirical results show that our new approach can achieve 30.6% line coverage (vs. 19.5% line coverage by Monkey) and 28.7% activity coverage (vs. 10.3% activity coverage by Monkey). Such improvements over Monkey are not only statistically substantial but also practically substantial given that the state-of-the-art research tools with advanced techniques cannot even outperform Monkey, as shown by Choudhary et al. [6].

More specifically, our approach leverages the UIAutomator[5] framework of Android to obtain all of the widgets that are enabled on a particular activity. With such information, our approach generates events exactly on the location of such widgets. Furthermore, our approach allows users to specify a weight for each event type on widgets. By allowing users to specify weights, our approach is able to perform weighted random selection to reduce redundant events. Additionally, our approach focuses on generating state-changing events by guiding the exploration to prefer widgets with higher likelihood to cause GUI-state changes.

This paper makes the following main contributions:

- The first industrial case study of applying Monkey on WeChat, a popular messenger app with over **762 million** monthly active users, and empirical findings on Monkey's limitations in an industrial setting.
- A new approach that addresses the major limitations of Monkey and accomplishes substantial code-coverage improvements over Monkey, along with empirical insights for future enhancements of both Monkey and our approach.

[4]http://a16z.com/2015/08/06/wechat-china-mobile-first/
[5]https://developer.android.com/topic/libraries/testing-support-library/index.html

Table 1: WeChat codebase statistics.

# of executable Java code lines:	610,629
# of Java classes:	8,425
# of Android activities:	607
# of C or C++ code lines:	∼40,000

2. BACKGROUND - WECHAT

Besides serving as a messenger app and social network, WeChat also contains additional functionalities found in apps such as PayPal, Yelp, Facebook, Uber, Amazon, etc. WeChat has even gradually evolved to be a platform for third parties to develop their official accounts (*e.g.,* light-weighted apps) running inside WeChat. Since WeChat contains many complicated features, it inevitably has a large code base as presented in Table 1 based on WeChat version 6.3.15.

When testing the WeChat Android client, we focus on Java code coverage, because the majority of the app's logic is implemented in Java, and its Java code is frequently changed between different versions of WeChat. We develop our own tool for measuring Java code coverage for two main reasons. First, it is desirable for us to have a tool that we can customize for various advanced testing features, such as measuring and comparing coverage information on only changed portions of the code between revisions. Second, existing coverage measurement tools such as Emma [2] are not able to handle large code bases such as WeChat's. In particular, the instrumentation performed by Emma (adding two methods into each class of the app under measurement) causes industrial-strength apps such as WeChat to reach the 64K-method limit after instrumentation.

Our coverage measurement tool collects (1) line coverage: the number of executed Java lines over the total number of executable Java lines; (2) activity coverage: the number of Android activities visited over the total number of Android activities.

3. STUDYING MONKEY ON WECHAT

Although there are many research tools [7, 3, 4, 5, 8, 10] to automatically generate inputs to test Android apps, none of these tools (including the six research tools studied by Choudhary et al. [6]) is usable on WeChat due to various applicability issues. In our study on testing WeChat, we focus on applying only Monkey. In particular, we set up Monkey to fire random events every 500 milliseconds such that it is long enough for WeChat to react and reach a stable state. We test WeChat version 6.3.15 on a Nexus 5 with Android OS version 5.1.1.

We run Monkey for 5 times independently. Each time we run Monkey for 18 hours with newly registered accounts. Most mobile apps require a correct user name and password (to be supplied in the correct fields) for the user to access the core functionalities of the apps. To enable Monkey to explore the functionalities of WeChat before and after login, we first configure Monkey to run for 2 hours without login. We then manually log in with a newly registered account for WeChat and then further run Monkey on WeChat for 16 hours.

In the default configuration, Monkey conducts system-wide exploration (e.g., it can open and explore all apps installed in the system). Such an exploration strategy reduces Monkey's effectiveness on WeChat, because most explorations conducted on other apps will not improve WeChat's activity or line coverage. Therefore, we configure Monkey to explore only apps whose package name is the same as the

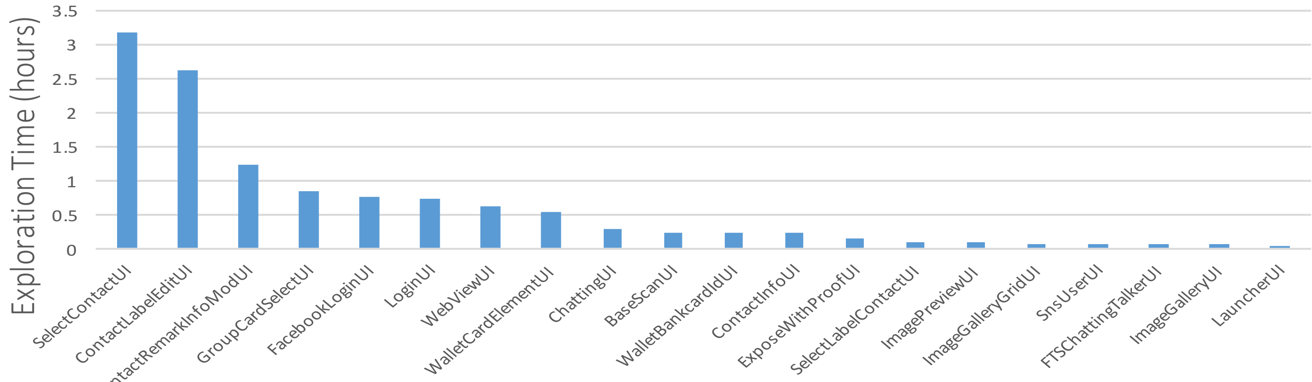


Figure 1: Top 20 activities based on the amount of exploration time that Monkey spends.

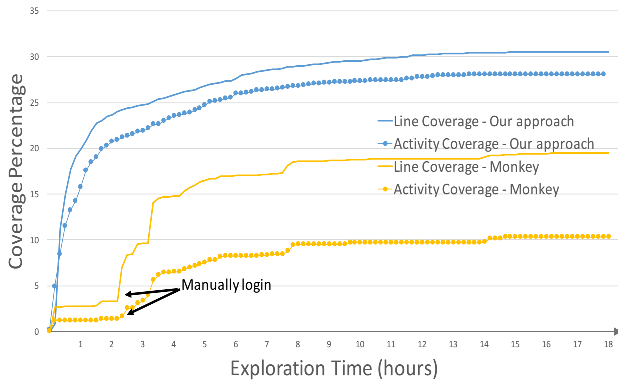


Figure 2: Line and activity coverage achieved by Monkey and our new approach.

WeChat package name. To understand the effects of inter-app communication, we conduct a manual study described in Section 5. Furthermore, we pin WeChat as the front window, which is supported by Android version 5.0 or higher, to prevent Monkey from exploring the system's StatusBar, which can be used to access system notification messages.

3.1 Coverage Results

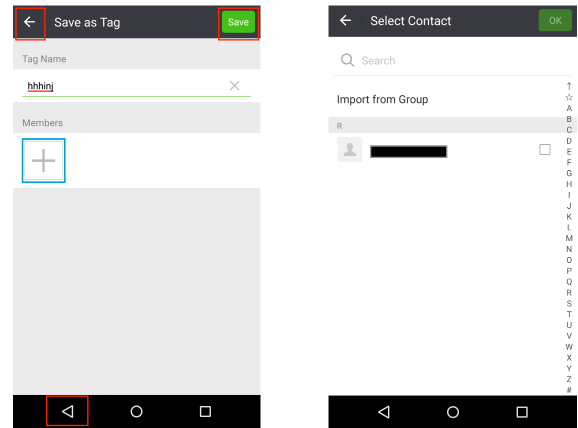
The lower two curves in Figure 2 shows the average line and activity coverage across all experiments achieved by Monkey. By the end of 18 hours, Monkey achieves on average 19.5% line coverage and 10.3% activity coverage.

We expected activity coverage to be higher than line coverage, because an activity can be thought of as a class, and class coverage is generally higher than line coverage. However, the achieved activity coverage is less than the line coverage.

> **Finding 1:** Monkey achieves both low line coverage and low activity coverage.

It is natural to expect that line coverage would keep increasing when activity coverage reaches its saturation point, because revisiting an explored activity could enhance the chance of exploring additional code logic that was not explored in previous visits of the same activity. However, the results in Figure 2 contradict such expectation because the line coverage remains constant when the activity coverage remains constant.

> **Finding 2:** Line coverage achieved by Monkey does not increase after activity coverage achieved by Monkey reaches its saturation point.



(a) contactLabelEdit activity (b) selectContactUI activity

Figure 3: Top 2 activities based on the amount of exploration time that Monkey spends.

3.2 Discussion

During Monkey's exploration, we repeatedly observe Monkey exploring the same screens for a long time. To further analyze this issue, we present the average time spent on each activity across all experiments as shown in Figure 1. The top 4 activities explored by Monkey constitutes 43.3% (7.8 out of 18.0 hours) of its exploration time.

> **Finding 3:** Monkey allocates a lopsided distribution of exploration time on each activity.

After closely examining top activities based on the amount of exploration time that Monkey spends, we identify the following two root causes that limit Monkey's exploration effectiveness.

Widget obliviousness. Since Monkey triggers events on random coordinates of a screen and has no knowledge of the location of widgets on a screen, Monkey generates many effect-free events (*e.g.*, events that do not trigger any functionality of WeChat and do not contribute to new line or activity coverage). For example, Figure 3b shows the activity that consumes the most exploration time. Unless Monkey generates events with coordinates at the three small-sized buttons (as marked with red rectangles), Monkey will continue to stay within this activity.

State obliviousness. We observe that Monkey explores the same two activities repeatedly without contributing to new code coverage. More specifically, the cycle begins with the activity in Figure 3a. Monkey clicks on the "+" button

Algorithm 1: Our exploration strategy.

```
Require: α ← assigned weight for event type on each widget
 1: procedure EXPLOREACTIVITY
 2:     widgets ← all widgets on current activity
 3:     time ← time spent on current activity
 4:     selectedWidget ← ∅
 5:     if time > totalTime * 0.4 then
 6:         Click RETURN button
 7:     else
 8:         unVisitedSet ← getUnVistedSet(widgets)
 9:         newActSet ← getNewActSet(widgets)
10:         refreshSet ← getFreshSet(widgets)
11:         if unVisitedSet ≠ ∅ then
12:             selectedWidget ← random from unVisitedSet
13:         else if newActSet ≠ ∅ then
14:             selectedWidget ← random from newActSet
15:         else if refreshSet ≠ ∅ then
16:             selectedWidget ← random from refreshSet
17:     if selectedWidget ≠ ∅ then
18:         chosenEvent ← random(α, selectedWidget)
19:         Fire chosenEvent
20:         Update time, totalTime, widgets
21:     else
22:         Click RETURN button
```

in this figure (marked with the blue rectangle) and navigates to the selectContactUI activity (Figure 3b). From the selectContactUI activity, Monkey eventually clicks the back arrow button to return to the contactLabelEditUI activity and the cycle restarts. Such repeated actions result in redundant explorations and occupy much of the exploration time.

4. ENHANCING MONKEY

To inherit the advantages of Monkey while addressing its two major limitations discussed in the preceding section, we develop a new approach incorporating two main strategies.

4.1 New Approach

Algorithm 1 shows the key algorithm of our approach. In particular, our approach incorporates two main strategies: widget awareness and state awareness with guided exploration.

Widget awareness. To alleviate Monkey's limitation of widget obliviousness, we leverage the UIAutomator framework of Android to obtain all the events (*e.g.,* short or long clicks) supported by each widget and perform only those events on the widgets. Our approach also allow users to specify a weight for each event type on each widget type. This mechanism allows our approach to use such predefined weights to perform weighted random selection to reduce many redundant events, as shown in Line 18 of Algorithm 1. For example, for the widget type of TextView, a user can assign 0.8 to a short click event and 0.2 to a long click event. With such predefined weights, when a TextView is selected, there is an 80% chance that our approach will perform a short click on it.

State awareness with guided exploration. To avoid repeatedly performing events without contributing to new line coverage, our approach focuses on generating events that may change the state. Our approach considers two states to be equivalent if the two states represent the same activity with the same number and type of widgets (the attribute values of the widgets can be different, e.g., the text in a TextView can be different). In particular, our approach represents a state as the mapping of an activity to the number and type of widgets that belong to this activity. Furthermore, our approach guides the exploration by selecting widgets with a higher likelihood to change the state. Our approach works by categorizing widgets on an activity into four categories:

- UnVisitedSet: Widgets that have not been visited before (typically events on these widgets have a higher chance of producing a new GUI state).
- NewActSet: Widgets that have been visited and at least one event enabled on these widgets caused our approach to go to another activity.
- RefreshSet: Widgets that have been visited before and none of the events on these widgets caused our approach to go to another activity but events on these widgets have created new states.
- DeadSet: Widgets that have been visited before and do not fall into the categories of NewActSet or RefreshSet.

After categorizing widgets, our approach selects widgets to explore based on the following priority: $UnVisitedSet > NewActSet > RefreshSet > DeadSet$. For each of the categories, our approach randomly picks a widget and performs weighted event selection as described earlier. If all widgets on the current activity are in the DeadSet, our approach will stop exploring the current activity, as shown in Line 22.

To further even the exploration time among activities, our approach records the time already spent on the current activity. If the total time performed on the current activity exceeds 40% of the total exploration time, our approach will trigger the "RETURN" button and go back to the previous activity, as shown in Line 6.

4.2 Coverage Results and Comparison

The tool that we develop for our approach supports automatic login with account information provided by the user of the tool. Thus, after our tool is started, it is no longer necessary for us to manually intervene. We run our tool five times and each time for 18.0 hours with the same basic setup as described in Section 3.

Figure 2 presents the average coverage results of our tool across all experiments and compares our approach with Monkey. As shown in Figure 2, our approach outperforms Monkey in both line coverage and activity coverage, covering 30.6% of the lines and 28.7% of the activities in WeChat. Said differently, our approach covers an additional 11.1% more lines and 18.4% more activities than Monkey does. These results reinforce our Findings 1 and 2 described in Section 3.

4.3 Discussion

Since our approach aims to balance the exploration time between activities, we expect exploration time to be evenly distributed among activities. As shown in Figure 4, the exploration time still has lopsided distribution among activities. Our approach spends 42.2% (7.6 out of 18.0 hours on average) of the exploration time in the top 4 activities on average. However, the activities that our approach spends the most time in are different than those that Monkey spends the most time in as shown in Figure 1.

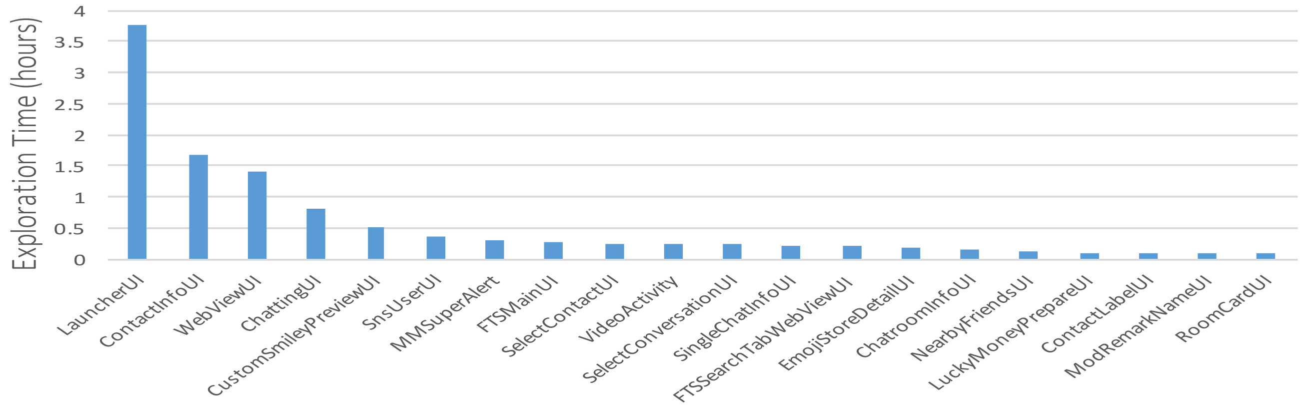


Figure 4: Top 20 activities based on the amount of exploration time that our approach spends.

Table 2: Events derived based on app domain knowledge.

Events	Condition
Scan valid QR code	Require a valid QR picture to be captured by the phone's camera.
Order taxi	Require valid address information.
Shake	Find friends through shaking the phone, which a tool cannot generate such shaking event.
Sign up	Require valid account information (*e.g.*, phone number or email address).
Inter-app communication	Require invoking other app to send an intent to WeChat.

Below are some top activities based on the amount of exploration time that our approach spends:

- *LauncherUI* is the root activity, namely the home screen, which allows users to access the majority of WeChat's features. Since this activity is the root entry point for the majority of WeChat's features, our approach constantly visits this activity during its exploration.

- *ContactInfoUI* and *ChattingUI* are easily accessible from other activities. One such activity is the LauncherUI. Since our approach constantly visits the LauncherUI, our approach also often visits the ContactInfoUI and ChattingUI activities.

- *WebViewUI* is the activity that displays web content to the user and is utilized by many activities since a significant number of tasks in WeChat display web content.

These popular activities generate many new states when our approach performs different events in them. However, these newly generated states do not significantly increase the line or activity coverage, since these states lead to many redundant exploration actions. For instance, Figure 5 presents a screenshot of WeChat under testing where our approach first tries to click the top "Call failed" TextField. The screen then adds another "Call failed" TextField. By introducing another "Call failed" TextField each time when a click is performed on these TextFields, our approach generates a new state but does not achieve additional coverage. This result indicates that such heuristic based state abstraction could sometimes be ineffective.

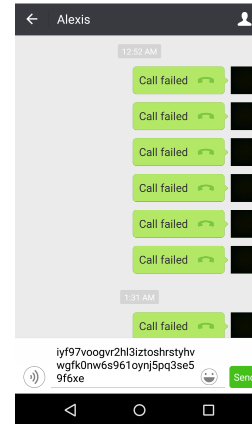


Figure 5: A chatting screen where our approach repeatedly fires click events on different TextView widgets without gaining coverage improvement.

5. HUMAN INTERVENTION

As described in Section 3, many features (such as inter-app communication) cannot be explored due to the limitations of our approach or the testing setting. In this section, with basic domain knowledge of WeChat (from the perspective of a typical user), we manually construct and perform events that our approach was incapable of in order to complement its capabilities. In addition, we intend to investigate whether it is feasible to provide human intervention based on basic knowledge of the app to further increase code coverage non-trivially.

Provide client-to-client interaction. We observe that WeChat sent messages to other friends' accounts and no other accounts responded to these messages. Because WeChat is a messaging-based app, and user interaction is its core functionality, after conducting automatic testing described in Section 4, we manually construct such client-to-client interaction by sending different message types (as shown in Figure 6) from other accounts to the account under test to trigger additional interactions. The line coverage increases slightly from 30.6% to 32.5% and the activity coverage increases from 28.7% to 29.3%. The main reason for such minor coverage increase is that the logic of processing client-to-client interaction is mainly implemented on the server side, instead of the client side. In summary, such human intervention may effectively help improve code coverage of the overall system (especially the server code) but not significantly to the client code.

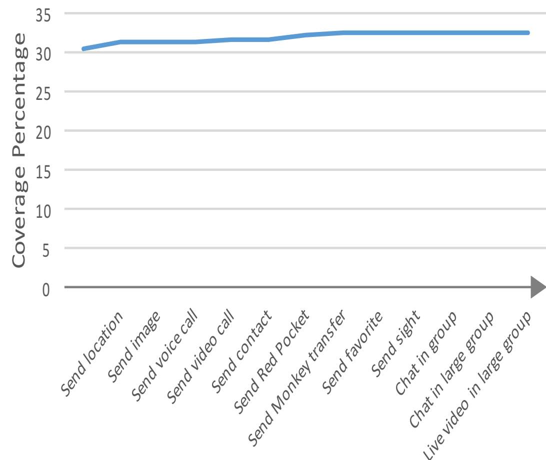


Figure 6: Line coverage statistic for client-to-client communication events performed in time sequence.

Provide additional app events. Based on our knowledge of WeChat's features, we construct a set of events that we believe would increase code coverage non-trivially, as listed in Table 2. Those events are also difficult for a tool to trigger because these events either require special external environment conditions or require valid inputs, as described in Table 2.

However, after performing those events, the line coverage increases unsubstantially from 32.5% to 33.3% and the activity coverage from 29.3% to 30.1% on average. We find that most of the triggered actions execute code that had already been covered previously, such as the WebViewUI as described in Section 4.3.

To cover more activities would require substantial effort from testers to understand the code base and its logic, and then construct appropriate event sequences to explore those not-covered code portions. Although WeChat's app logic is properly documented, it is still labor-intensive to derive tests from the documentation, especially since WeChat updates its app logic frequently to incorporate additional features.

Finding 4: It is challenging for human with basic knowledge of the app domain to derive event sequences to further improve code coverage of WeChat's client code.

6. FUTURE WORK

In this section, we discuss the limitations and planned features of our new approach as our future work.

Incorporating fine-grained code analysis. As motivated by Finding 2, exploring different features or activities during testing could effectively help increase line coverage. Conducting code analysis can help existing tools achieve higher code coverage; for example, such analysis (*e.g.,* constructing the activity transition graph [9]) can guide the tools to explore not-covered activities as exploration targets. Related previous work [4] has proposed techniques of using taint analysis to find widgets that can trigger the target activity transition. However, triggering such activity transition is based on approximated information. For example, clicking those widgets does not necessarily trigger the target activity transition unless a certain condition (*e.g.,* specific events are earlier performed) is satisfied. Fine-grained code analysis could offer additional help here.

Reusing exploration paths. In practice, existing tools test the app under test with multiple settings (*e.g.,* different Android OS, smartphone, and app version) and each run may produce exploration paths that cover different app features. Extracting relevant paths from past explored paths and reusing them for future exploration could help boost existing tools' effectiveness.

Incorporating new evaluation metrics. It is necessary to evaluate existing tools beyond only on code coverage. For example, a tool may excel in detecting certain types of program failures caused by Application Not Responding (ANR) or improper inter-app communication.

7. CONCLUSION

In this paper, we have presented the first industrial case study of applying Monkey on WeChat, a highly popular messenger app with over 762 million monthly active users, and reported empirical findings on Monkey's limitations in such industrial setting. We have also presented our new approach to address the major limitations of Monkey and accomplish substantial code-coverage improvements over Monkey, along with empirical insights for future enhancements to both Monkey and our approach.

8. ACKNOWLEDGMENTS

We thank Zhenyu Lei and Haibing Zheng for helping set up testing accounts and configurations. The work is supported in part by NSF under grants no. CCF-1409423, CNS-1434582, CCF-1434596, CNS-1513939, CNS-1564274.

9. REFERENCES

[1] Android 64k method limit. https://developer.android.com/studio/build/multidex.html.

[2] Emma: a free Java code-coverage tool. http://emma.sourceforge.net/.

[3] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE'12*, pages 599–609.

[4] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA'13*, pages 641–660.

[5] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA'13*, pages 623–640.

[6] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *ASE'15*, pages 429–440.

[7] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *FSE'13*, pages 224–234.

[8] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of Android apps. In *FSE'14*, pages 599–609.

[9] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *FSE'12*, pages 658–668.

[10] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE'13*, pages 250–265.