Automated Test Input Generation for Android: Towards Getting There in an Industrial Case

Haibing Zheng[1] Dengfeng Li[2] Beihai Liang[1] Xia Zeng[1] Wujie Zheng[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]{mattzheng,gavinliang,xiazeng,wujiezheng,yuetangdeng}@tencent.com, [2]{dli46,winglam2,weiyang3,taoxie}@illinois.edu

***Abstract*—Monkey, a random testing tool from Google, has been popularly used in industrial practices for automatic test input generation for Android due to its applicability to a variety of application settings, *e.g.,* ease of use and compatibility with different Android platforms. Recently, Monkey has been under the spotlight of the research community: recent studies found out that none of the studied tools from the academia were actually better than Monkey when applied on a set of open source Android apps. Our recent efforts performed the first case study of applying Monkey on WeChat, a popular messenger app with over 800 million monthly active users, and revealed many limitations of Monkey along with developing our improved approach to alleviate some of these limitations. In this paper, we explore two optimization techniques to improve the effectiveness and efficiency of our previous approach. We also conduct manual categorization of not-covered activities and two automatic coverage-analysis techniques to provide insightful information about the not-covered code entities. Lastly, we present findings of our empirical studies of conducting automatic random testing on WeChat with the preceding techniques.**

I. Introduction

Recently Choudhary et al. [12] asked the question "are we there yet?" in terms of having good-enough tools to automatically generate inputs to test Android apps. They conducted an empirical study on publicly available tools that can automatically generate inputs to test Android apps. These tools included six test input generation tools [14], [9], [10], [11], [15], [19] from the academia, in short as academic tools. In addition to the six academic tools, the study also considered Monkey[1], an open source tool from Google. Monkey is one of the most widely used tools of this category under industrial settings, due to its applicability to a variety of application settings, *e.g.,* ease of use and compatibility with different Android platforms. Monkey is considered to be a relatively simplistic tool because Monkey triggers random events on random coordinates of a screen. Although the six academic tools use more sophisticated techniques than Monkey, Choudhary et al. found Monkey to be the winner among the tools under their study, since it achieves, on average, the best coverage, it can report the largest number of failures, it is easy to use, and it works for various platforms.

In our recent efforts [20], we extended Choudhary et al.'s study [12] in two important ways. First, we performed the first case study of applying Monkey on an industrial app instead of some relatively simplistic, open-source apps. Our case study revealed many limitations of Monkey. Second, we developed an improved approach that addresses major limitations of Monkey and demonstrated how our approach accomplishes substantial code-coverage improvements over Monkey.

In particular, our recent efforts studied the effectiveness and limitations of Monkey when testing 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 **800 million** (2016 Quarter 2) monthly active users[3]. In fact, WeChat has evolved to be well beyond a messenger app: it also supports many functionalities such as banking and shopping, and serves as a platform for third parties to develop their own apps[4].

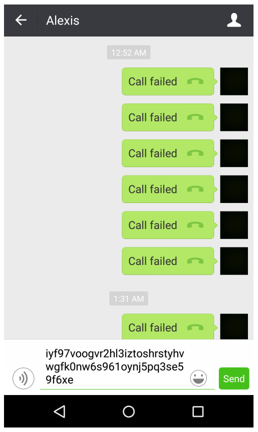
Such recent efforts were just the beginning of the journey started by the industry-academia partnership formed by the authors of this paper, consisting of industrial practitioners from Tencent, Inc. and university researchers from the University of Illinois at Urbana-Champaign. The goal of such partnership is to significantly improve both the state of the art and the state of the practice in the area of automated test input generation for Android, and produce high industrial impact beyond/besides impacting the academic research community in this area. Accomplishing this goal requires continuous dedicated efforts to gain improved understanding on "what does work and what does not work" and develop practical solutions to tackle challenges faced when transferring and deploying tools in real industrial practices.

In this impact-pursuit journey, we extend our recent efforts [20] by exploring various optimization techniques for random testing tools (such as Monkey and our previous approach [20] extended from Monkey) through empirical studies in this paper. More specifically, we optimize our previous approach by developing two techniques: the Repetition-Avoidance technique and the Firing-Speedup technique.

The Repetition-Avoidance technique aims to keep track of repeated exploration patterns that do not increase coverage and avoid such repetition during random exploration of the event space. Random testing tools such as our previous approach may keep track of whether a just-fired event changes the GUI

[1]https://developer.android.com/studio/test/monkey.html

[2]https://www.wechat.com
[3]https://www.statista.com/statistics/255778/number-of-active-wechat-messenger-accounts/
[4]http://a16z.com/2015/08/06/wechat-china-mobile-first/



Call failed
Call failed
Call failed
Call failed
Call failed
Call failed

Call failed

iyf97voogvr2hl3iztoshrstyhv
wgfk0nw6s961oynj5pq3se5
9f6xe

Fig. 1: A chatting screen where our previous approach [20] repeatedly fires click events on different TextView widgets without gaining coverage improvement.

elements of the screen or not. However, such mechanism is at times ineffective at testing an app. For instance, Figure 1 presents a screenshot of WeChat under test where our previous 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 a click is performed on these TextFields, our previous approach detects that GUI elements have changed, and therefore it continues exploration without the possibility of achieving new code coverage.

The Firing-Speedup technique aims to reduce the execution runtime for firing an event. In this way, within the same amount of time allocated for applying random testing tools, the tools can fire a much higher number of events compared with not using the Firing-Speedup technique. In our empirical observation (with more details described in later parts of this paper), our previous approach enhanced with the Firing-Speedup technique, namely the speed-optimized approach, can achieve the same code coverage as our previous approach using only 3 hours in contrast to 12 hours used by our previous approach. Note that neither approaches have reached their saturation point at 12 hours, and given infinite time, both approaches would achieve the same code coverage. When comparing our speed-optimized approach to the previous approach at 12 hours, our speed-optimized approach achieves approximately 6% higher code coverage. In summary, we find that our speed-optimized approach runs faster and more effectively.

In addition to the two optimization techniques, in this paper, we also conduct categorization of coverage results to inform developers what most common categories of activities are among those not-covered activities. In particular, we conduct a study on the not-covered activities of WeChat and identify seven major categories of not-covered activities. The category of Dead Activity is the most common category, accounting for 39% of the not-covered activities. An activity is categorized as a Dead Activity if the app usage data from WeChat's active users and our automatic testing tool never cover that activity. Our manual inspection of these activities finds that these activities often contain implementations of an outdated feature or a feature soon to be (but not yet) released. From this study's findings, app developers or testers can gain some insights about the not-covered activities such as which activities are in the Dead Activity category so that testing approaches can exclude them. The study results also inform developers about how significant not-covered activities may be. The impact of not covering activities of the Dead Activity category may not significantly impact the confidence that developers will have on their testing efforts. However, the second most popular category of not-covered activities (not covering activities whose coverage requires certain specific account states) should further motivate developers to spend additional resources during their testing efforts to cover such activities.

Aside from our categorization of coverage results, we also explore two automatic coverage-analysis techniques to produce insightful information about the not-covered activities: substring hole analysis and activity-transition-graph (ATG) analysis. Substring hole analysis [7], [8] automatically generates the list of keywords most frequent in the names of the not-covered activities. In particular, substring hole analysis splits a string into substrings based on a specified delimiter. Our use of substring hole analysis works by splitting activity names by capitalized characters and counting the occurrences of the keywords. The results from our substring hole analysis on the not-covered activities suggest that manual efforts or automated tools should aim to generate more tests that use the WeChat wallet to pay for things in order to maximize their chance to increase code coverage.

The ATG analysis constructs the activity transition graph (ATG) of the app under test to generate a list of activities that can lead to the not-covered activities. We leverage GATOR [18], a program-analysis toolkit for Android, to generate the ATG and investigate how the use of the generated list can help developers cover the not-covered activities.

In summary, this paper makes the following main contributions:

- Optimization techniques to improve the effectiveness and efficiency of automatic random testing for Android.
- Manual categorization of not-covered activities and coverage analysis (substring hole analysis and ATG analysis) to provide insightful information about not-covered activities.
- Empirical studies of conducting automatic random testing on WeChat with the preceding techniques.

In the rest of this paper, we present background information in Section II. We illustrate the testing methodology used for our study in Section III. We present our two optimization techniques in Section IV. We illustrate our categorization of not-covered activities and our coverage analysis in Sections V and VI, respectively. We discuss related work in Section VII and conclude in Section VIII.

TABLE I: 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

II. Background

A. WeChat

WeChat was first released in 2011 by Tencent, Inc. Since then WeChat has grown to be not only a messenger app and social network, but also a multi-function app containing many of the functionalities found in popular apps such as PayPal, Yelp, Facebook, Uber, and Amazon. WeChat has even gradually evolved to be a platform for third parties to develop their official accounts, i.e., light-weight apps, running inside WeChat. One example of WeChat's functionalities that may not be obvious is when eating out with a group of friends, one can use WeChat to split the check by sending a QR code out to everyone, each of whom can then automatically pay for their portion of the bill with the tap of a button inside WeChat. Other use cases of WeChat include but are not limited to buying movie tickets, calling a cab, online shopping, counting the number of footsteps taken, uploading and sharing photos, ordering delivery, reading news, and numerous other use cases.

Since WeChat contains many complicated features, it inevitably has a large code base as shown in Table I based on WeChat version 6.3.15.

B. Automatic Test Input Generation for WeChat

Our previous study [20] shows that Monkey, an Android random testing tool from Google, achieves surprisingly low line or activity coverage for testing industrial apps such as WeChat. The main reasons 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.

We developed our previous approach such that it inherits the high applicability of Monkey while addressing its empirically-observed limitations. In particular, our previous 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 UI Automator[5] APIs 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. The UI Automator APIs enable us to look up a UI component by using the displayed text or content description. Our previous approach also allows users to specify a weight for each event type on each widget type. This mechanism allows our previous approach to use such predefined weights to perform weighted random selection to reduce many redundant events.

[5]https://google.github.io/android-testing-support-library/docs/uiautomator/

State awareness with guided exploration. To avoid repeatedly performing events without contributing to new line coverage, our previous approach focuses on generating events that may change the state. Our previous 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 previous approach represents a state as the mapping of an activity to the number and type of widgets that belong to this activity. Furthermore, our previous approach guides the exploration by selecting widgets with a higher likelihood to change the state.

III. Testing Methodology

A. Coverage measurement

When testing the WeChat Android app, 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 use a tool developed in our previous work [20] for measuring code coverage for Java. Using our own coverage measurement tool offers us two major advantages. First, it allows us to customize the tool 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.

B. Testing setup

We conduct our testing experiments with WeChat version 6.3.15 on an OPPO R9 device running Android OS version 5.1.1. We run our previous approach and our new optimized approach for 12 hours, separately, with newly registered accounts on live servers (i.e., the ones used by the broad user base). Note that there are thousands of micro-services running on tens of thousands of backend servers across a few data centers. Since our approaches run on live servers, there is a chance that our approaches could have added nearby people as friends and potentially sent money to nearby people during testing. Therefore, for the 12 hours that we run our approaches, we purposefully do not test any financial-related features of WeChat (e.g., not manually bundling a bank card with a test account). Later in Section VI-A, we describe our lesson learned on the need of testing financial-related features to cover many not-covered activities. In order to test such features, we create a mock server, which contains only testing accounts. We then run our new optimized approach on the mock server, and measure the coverage improvement.

IV. Optimization Techniques

In this section, we first present the limitations of our previous approach and then two new optimization techniques that we develop to improve our previous approach. Lastly, we empirically evaluate the previous and optimized approaches by measuring the code coverage achieved by the two approaches.

A. *Repetition-Avoidance Technique*

During exploration, we repeatedly observe that our previous approach explores newly generated states for a long time. However, these newly generated states do not significantly increase the line or activity coverage, since these states lead to many redundant exploration actions. Figure 1 shows an example of these redundant exploration actions. In the example, our approach first tries to click the top “Call failed” TextField. The app then adds another “Call failed” TextField. Since the GUI elements on the screen are changed, our previous approach considers that such change leads to a new state even when such new state leads to redundant explorations that do not achieve any additional coverage.

To avoid redundant explorations, our repetition-avoidance technique allows developers to specify what buttons should not be clicked. In particular, when our technique generates events for an activity, our technique checks to see whether a generated event is one on a button specified by the developers. If so, then the event is discarded and another event will be generated; otherwise, the event is triggered. In the example shown in Figure 1, a developer can specify that a “Call failed” TextField should not be clicked on. When our technique generates events for this example, our technique checks each generated event to see whether it is on a “Call failed” TextField and if so, the event is discarded and another event will be generated.

B. *Firing-Speedup Technique*

Our firing-speedup technique reduces the runtime cost of our previous approach by being faster at picking GUI widgets and firing events on them. This improvement enables our approach to achieve coverage increments faster. Our previous approach performs the following steps to pick a GUI widget and fire an event on the widget: (1) get the GUI hierarchy tree from the UI Automator and traverse the GUI hierarchy tree to get all of the widget objects, which contain widget-attribute information such as coordinates, text, and supported event types (e.g., short and long clicks); (2) pick one widget from the GUI hierarchy tree and a supported event type, and pass them to the UI Automator. The UI Automator also needs to traverse the GUI hierarchy tree to verify that the input GUI widget and event are valid before it can fire the event. Therefore, our previous approach needs to traverse the GUI hierarchy tree twice. To eliminate the need for the second GUI hierarchy traversal by the UI Automator, we fire the event on the widget through the Android Debug Bridge (ADB) (instead of the UI Automator) immediately after the first GUI

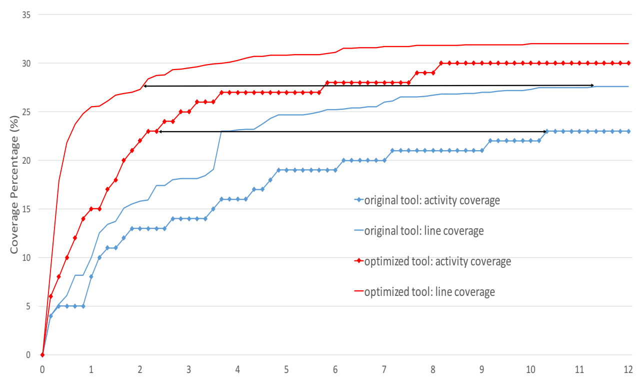


Fig. 2: Comparison of the line/activity coverage achieved by our optimized and previous approaches.

TABLE II: Comparison of the numbers of events generated by our optimized and previous approaches.

Time (minutes)	Number of events fired	
	Optimized approach	Previous approach
30	1050	509
60	1935	1023
90	2822	1517
120	3706	1988
150	4685	2459
180	5772	2957
210	6754	3428
240	7585	3868

hierarchy traversal. For event types such as a long click, which cannot be fired by ADB, we continue to leverage the UI Automator to fire these events. Our firing-speedup technique brings significant reduction of exploration time compared to our previous approach.

C. *Results*

Table II lists the numbers of events that are generated by our optimized and previous approaches. As shown in Table II, our optimized approach generates twice as many events as our previous approach within the same time period. Figure 2 presents the coverage result achieved by our optimized and previous approaches. The x-axis shows the number of hours spent on exploration and the y-axis shows the coverage percentage. As shown in Figure 2, our optimized approach (the upper two lines) outperforms our previous approach (the lower two lines) in both line and activity coverage. Specifically, our optimized approach covers an additional 6.0% more lines and 6.0% more activities than our previous approach. Also, our optimized approach has faster coverage increments than our previous approach. The two black, horizontal lines in Figure 2 indicate that our optimized approach needs only 2 hours to achieve the coverage achieved in 12 hours by our previous approach.

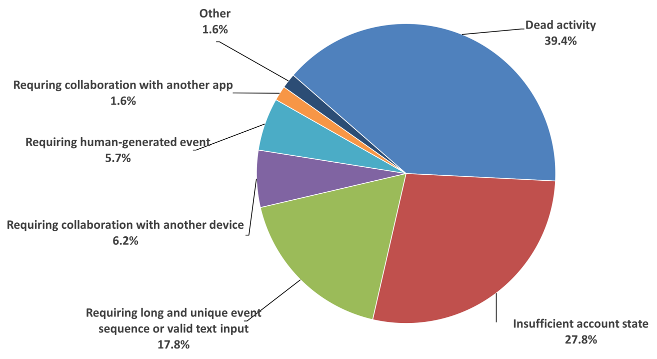


Fig. 3: Classification of not-covered activities.

V. Categorization of Not-covered Activities

In this section, we present the categorization of WeChat's activities that have not been covered by our optimized approach, and we further describe their root causes and implications, as shown in Table III.

We manually investigate the source code of those not-covered activities and categorize them into 7 major categories and 12 minor categories based on the conditions needed to trigger these activities. The results are shown in Figure 3.

A. Dead Activity

We categorize an activity as a Dead Activity if it has never been covered by an active user of WeChat or by our optimized approach. We collect app usage data for one day from 200+ million users of the same WeChat version. Such app usage data contains all activities visited by users on that particular day. In other words, we aggregate all visited activities across 200+ million users of the same WeChat version and obtain a list of activities that have been covered by the active users. If an activity in WeChat is not covered by an active user and is also not covered by our optimized approach, then we categorize it as a Dead Activity.

Surprisingly, we find that 39.4% (173 out of 439) of the not-covered activities are in the category of Dead Activity. After examining the source code of those activities, we find two major reasons for dead activities in WeChat as below.

Old implementations. An activity may become out-dated if a new implementation of the same feature has been deployed. For example, older versions of WeChat use the native WebView in Android to implement their web features. However, developers have already replaced such implementation with a newer version using HTML5 while the old implementation continues to reside in WeChat. Our optimized approach and active users can no longer visit the old implementation anymore. On the other hand, it is risky for developers to refactor the code base and remove those old implementations for two main reasons. First, code refactoring may cause device-compatibility issues since it is challenging to test all kinds of devices and confirm that removing old implementations will not negatively affect some devices. Second, developers may still use those old implementations for future feature development.

Unreleased and hard-to-cover features. We find that certain features in WeChat are not yet released to the public and by default are disabled because the backend server is not available yet to support such features. Such cases cause some activities to be invisible (i.e., infeasible to cover) to active users and our optimized approach.

Implication. It is important for developers to know the list of dead activities in advance so that they can remove such activities from the not-covered activity list when evaluating the effectiveness of their testing approaches.

B. Insufficient Account States

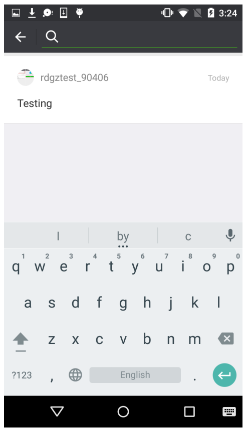
If the condition to reach a particular activity cannot be satisfied by directly providing an initial account state for a testing account, then we categorize this activity into the category of insufficient account states. For example, to test the email activity in WeChat, developers are required to provide an account that has the email feature enabled. We find that 27.8% (122 out of 439) of the not-covered activities need proper account configurations so that testing approaches such as our optimized approach can cover those activities. There are four primary settings for account states as below.

Requiring financial information. We find that 9.6% (42 out of 439) of the not-covered activities are related to the financial features of WeChat (i.e., using WeChat's wallet or bank payment). To cover those activities, our optimized approach would need an account that has the proper financial information (e.g., a testing account with at least one bank card with non-zero balance). Setting up our optimized approach to test those activities is challenging because those activities need to perform transactions between different testing accounts (e.g., sending money to people from one's friend list). Moreover, some activities depend on the results of transactions (e.g., a transaction is rejected by a bank). Thus, it is challenging to generate proper testing accounts. In addition, if generated tests are executed against a testing account on a live server, the testing account may potentially send money to another non-testing account (i.e., an active user).

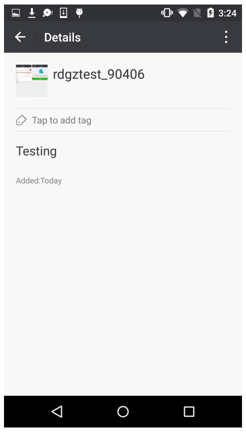
Implication. In practice, to test financial features, developers need to (1) preset financial information (e.g., bank card), and (2) combine manual testing and automated testing using an offline or mock server to test those activities.

Requiring account-history content. We find that 7.5% (33 out of 439) of the not-covered activities need a testing account with some history content for our optimized approach to effectively explore these activities. For example, Figure 4 presents an activity for searching a user's saved favorite content. If the account does not save a favorite content from before, the subsequent activity for showing the detailed content cannot be visited.

Implication. Developers should add initial seed data to testing accounts or reuse some of their previous testing accounts that contain some app usage history.



(a) Activity for searching saved favorite history



(b) Activity for showing details of searching result

Fig. 4: An example where testing accounts with saved favorites are needed in order to visit the searching result activity.

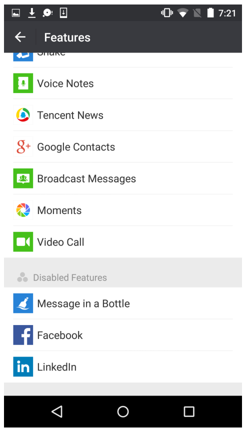


Fig. 5: Example of features in WeChat that are disabled by default.

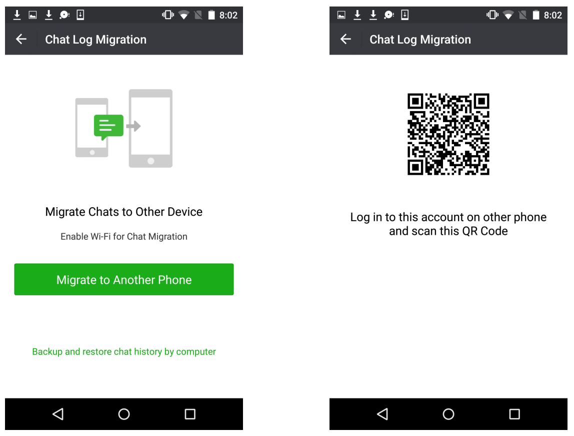
Requiring different account types. We find that 7.3% (32 out of 439) of the not-covered activities are not covered because our optimized approach supports only one way of login: login with a WeChat account. Besides supporting WeChat accounts, WeChat also supports sign-up and login to various third-party accounts, such as Facebook and Tencent QQ accounts. The support for multiple account types leads to different activities for sign-up and login features. Also, an account registered within China has different features or uses different implementations (i.e., different activities) compared to an account registered outside of China. However, we evaluate our optimized approach using only a WeChat account registered within China.
Implication. Testing approaches should support using different account types to log in instead of using only one account type.
Requiring enabled feature. We find that 3.4% (15 out of 439) of the not-covered activities are not covered because features related to those activities are not enabled. Figure 5 presents an example list of features that are not enabled by default. To cover the activities in this category, developers would need to either (1) manually enable those features for a testing account on their backend server or (2) guide testing tools to visit WeChat's setting activity and click the enable button for each feature, as shown in Figure 5.
Implication. Developers need to create a group of testing accounts such that each testing account has a different initial setting (i.e., accounts with different enabled features).

C. Requiring long and unique event sequence or valid text input

We categorize a not-covered activity to this category if reaching the activity requires following a unique path of activity transitions and such path consists of more than five activity transitions from the root activity (i.e., the Launcher activity) to the target activity or requires filling in correct text inputs. This category includes 17.8% (78 out of 439) of the not-covered activities, including two sub-categories as below.
Requiring long and unique event sequence. 16.4% (72 out of 439) of the not-covered activities requires one unique path from the root activity (i.e., the Launcher activity) to the target activity, and such unique path is generally a long sequence, i.e., requiring at least five activity transitions. It is difficult for a random testing approach to cover those activities. For example, to send a broadcast message to many friends at once, a testing approach needs to go through the following path of activity transitions: $LauncherUI- > Setting- > General- > FeatureSettings- > GroupMessaging- > BroadcastsMessages$.
Implication. It is suggested that testing approaches support developer-specified rules so that these approaches can utilize such rules to more easily navigate to not-covered activities in this category.
Requiring valid text input. We find that 1.4% (6 out of 439) of the not-covered activities are not covered because they require the optimized approach to provide correct text input. For example, to explicitly mention a group member in a group chatting room, the user is required to type in the character "@", and then a window will pop up to allow the user to select the group members.
Implication. It is suggested that testing approaches support taking advantage of predefined text inputs (e.g., the "@" character) for activities in this category.

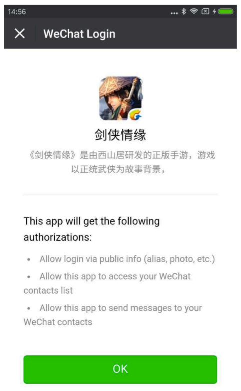
D. Requiring collaboration with another device

We find that 6.2% (27 out of 439) of the not-covered activities are not covered because they require our testing approach to provide collaboration between WeChat and another device. For example, WeChat allows users to migrate or backup their chatting history to another device, as shown in Figure 6. To test data migration to another phone requires the other phone


8:02
Chat Log Migration
Migrate Chats to Other Device
Enable Wi-Fi for Chat Migration
Migrate to Another Phone
Backup and restore chat history by computer
8:02
Chat Log Migration
Log in to this account on other phone
and scan this QR Code

(a) Migrate or backup data to another device **(b)** Scan QR code to migrate data to another phone

Fig. 6: Data migration and backup in WeChat require collaboration between different devices.



《剑侠情缘》是由西山居研发的正版手游，游戏以正统武侠为故事背景，

This app will get the following authorizations:

- Allow login via public info (alias, photo, etc.)
- Allow this app to access your WeChat contacts list
- Allow this app to send messages to your WeChat contacts

Fig. 7: Another app sends inter-component communication to WeChat for login authorization.

to scan the QR code as shown in Figure 6b. To test data backup to a desktop requires the desktop-version application and the phone to be connected to the same Wi-Fi network.

Implication. Support is suggested for allowing testing tools on different devices or platforms to be able to communicate with each other and work cooperatively for covering these activities.

E. Requiring human-generated event

Except for those not-covered activities categorized into the preceding categories, if covering a not-covered activity requires events that cannot be generated by our optimized approach, we categorize such activities into this category. This category includes 5.7% (25 out of 439) of the not-covered activities, including two sub-categories as below.

Requiring biometric information. We find that 3.4% (15 out of 439) of the not-covered activities are not covered because they require correct events for biometric information, such as using the voice message or fingerprint features in WeChat.

Implication. It is suggested that testing approaches should be able to recognize activities requiring biometric information, and cooperate with developers to provide or mock such biometric information during testing.

Requiring system event. We find that 2.3% (10 out of 439) of the not-covered activities are not covered because they require a particular system event, such as connecting to Wi-Fi. Our optimized approach does not support directly sending commands to the Android system to generate system events.

Implication. There is a need for testing approaches to provide support to generate system events. One possible solution is to instrument an Android system such that it allows testing approaches to generate system events related to the app under test [14].

F. Requiring collaboration with another app

We find that 1.6% (7 out of 439) of the not-covered activities are not covered because they require inter-component communication (ICC) between WeChat and another app. For example, Figure 7 presents the activity where another app tries to get the user's WeChat account information as login authorization.

Implication. An instrumented Android system is suggested to be used for testing. By checking WeChat's broadcast receivers, an instrumented Android system can extract what ICC message WeChat is listening to, and allow testing approaches such as our optimized approach to automatically generate such ICC messages [14].

G. Other

The remaining 1.6% (7 out of 439) of our activities are classified in this category. The patterns of these not-covered activities are not representative and do not occur frequently. For example, one activity categorized to this category is an activity to support the walkie-talkie feature of WeChat specifically for the current version to talk to an older version of WeChat.

Implication. It is suggested that testing approaches support developer-specified rules so that these approaches can utilize such rules to more easily navigate to the not-covered activities in this category.

VI. Coverage Analysis

This section presents our two coverage-analysis techniques for helping developers more easily identify the steps needed to explore not-covered activities.

A. Substring Hole Analysis

In order to provide developers additional insights on what activities are not covered, we develop a tool for substring hole analysis [7], [8] to analyze the names of such activities. A substring hole is a set of activity names that have a common substring, and at least one activity from this set is a not-covered activity. The common substring is used as

TABLE III: Categorization of not-covered activities and their implications.

Category	Subcategory	Percentage	Description	Implication
Dead activity	Dead activity	39.4%	No testing approach can cover these activities because (1) activities of a feature become outdated if a new implementation of the same feature has been deployed; or (2) activities are related to some features not yet open to the public and thus disabled.	Developers shall know about these dead activities in order to remove them from the not-covered activities when assessing the effectiveness of their testing approaches.
Insufficient account state	Requiring financial information	9.6%	Testing accounts by default are not bundled with a bank card, and thus cannot be used to cover these activities related to financial transactions (e.g., WeChat wallet and bank payment). And conducting automated testing of wallet/payment features on a deployed server can incur financial transactions with other non-testing accounts (i.e., active users).	Developers shall (1) preset financial information (e.g., bank card), and (2) use an offline or mock server to test these activities.
	Requiring account-history content	7.5%	Testing accounts by default do not have history content (e.g., saved favorite messages), and thus cannot be used to cover these activities.	Developers shall add initial seed data to testing accounts or reuse some of the previous testing accounts that contain history content.
	Requiring different account types	7.3%	Testing approaches support login of only one account type, and thus cannot cover these activities related to login features of other different account types.	Testing approaches shall support using different account types to log in instead of using only one account type.
	Requiring enabled feature	3.4%	Testing accounts by default have the same initial configurations where certain features are disabled; thus, activities related to these features cannot be covered.	Developers shall create a group of testing accounts, each of which shall have a different initial configuration for enabling a different feature.
Requiring long and unique event sequence or valid text input	Requiring long and unique event sequence	16.4%	Testing approaches cannot effectively generate a specific path (with >=5 activity transitions) from the root activity to the target activity, and thus cannot cover the target activity.	Testing approaches shall support developer-specified rules to guide them to more easily navigate to these not-covered activities.
	Requiring valid text input	1.4%	Testing approaches do not provide direct support to generate valid text inputs, and thus cannot cover these activities.	Testing approaches shall be provided with a set of predefined text inputs for these activities.
Requiring collaboration with another device	Requiring collaboration with another device	6.2%	Testing approaches do not allow the device used for testing to interact with another device, and thus cannot cover these activities.	Testing tools on different devices or platforms shall be able to communicate with each other and work cooperatively to cover these activities.
Requiring human-generated event	Requiring biometric information	3.4%	Testing approaches do not provide biometric information, such as voice message and fingerprint, to cover these activities.	Testing approaches should be able to recognize activities that require biometric information and cooperate with developers to attain such biometric information.
	Requiring system event	2.3%	Testing approaches do not generate system events such as connecting to and disconnecting from Wi-Fi, and thus cannot cover these activities.	There is a need for testing approaches to provide support to generate system events [14].
Requiring collaboration with another app	Requiring collaboration with another app	1.6%	Testing approaches do not cause proper inter-component communication (ICC) messages to be sent from another app, and thus do not cover these activities	An instrumented Android system can be used for allowing to extract what ICC messages WeChat is listening to by checking WeChat's broadcast receivers, and thus enabling testing approaches to generate such ICC messages to cover these activities [14].
Other	Other	1.6%	The patterns of these not-covered activities are not representative and do not occur frequently.	Testing approaches shall support developer-specified rules to guide them to more easily navigate to these not-covered activities.

the identifier for the substring hole. To detect substring holes, our tool splits activity names (by capitalized characters) into keywords and then counts the occurrences of the keywords. For example, for an activity named FindCreditCardUI, our tool identifies four keywords: find, credit, card, and ui.

For our study, we first apply our tool on the name set of covered activities and not-covered activities. We generate this set by applying our optimized approach on WeChat for 12 hours. As shown in Table IV for top identified substring holes, the most frequent keyword (i.e., common substring) in the names of the not-covered activities is "ui", with 398 occurrences among the not-covered-activity names and in total 574 occurrences among all activity names (i.e., being present in the names of about 95% activities). The second most frequent keyword in the names of the not-covered activities is "wallet", with 61 occurrences among the not-covered-activity names and in total 71 occurrences among all activity names. This finding indicates that both a high percentage (85.9%)

TABLE IV: Substring holes after applying our approach for 12 hours.

Substring	Uncovered / Total
ui	398 / 574 (69.3%)
wallet	61 / 71 (85.9%)
detail	27 / 37 (73.0%)
pay	26 / 27 (96.3%)
card	22 / 24 (91.7%)

and a high number (61) of wallet-related activities are not covered. Since a high number of wallet-related activities are not covered and one of WeChat's main functionalities of wallet is to allow users to pay for goods or services with a bank card, it comes as no surprise that the fourth and fifth most frequent keywords in the names of the not-covered activities are "pay" and "card", respectively. The reason for the high number of not-covered pay/card-related activities is that our optimized approach is applied on live servers and we purposefully do not test any financial-related features of WeChat (e.g., paying someone with a bank card) because there is a chance that our approach could send money to nearby people during testing.

To further attempt to cover more wallet-related not-covered activities, we derive 1766 wallet/payment-related test cases from documented app logic to manually test not-covered activities. It takes one developer 48 hours to perform the 1766 test cases. By performing the 1766 test cases, we reduce the substring hole of "wallet" to be about 22.5% (16 / 71) from 85.9% (61 / 71). Our results show that by manually testing wallet-related activities as suggested by our substring hole analysis, we can significantly cover more activities that are not covered by our optimized approach.

B. *Critical-Activity Analysis*

We also investigate the pattern of not-covered activities using an activity transition graph (ATG) with the coverage information. ATG is a directed graph in which each node denotes an Android activity, and each edge denotes an operation of changing the currently-visible window from the source activity to the target activity. The purpose of such investigation is to locate the critical bottlenecks of app exploration. We define a *critical activity* as a not-covered activity that is the only predecessor of some other not-covered activities in the ATG. Since covering a critical activity could enable the testing tool to likely cover its successor activities, locating the critical activity is crucial to improving the app-exploration strategy and consequently, improving the code coverage.

To carry out the investigation, we leverage GATOR [18] to obtain the ATG. In particular, GATOR constructs a Window Transition Graph (WTG). A WTG is similar to an ATG except that nodes in a WTG also include windows (e.g., dialog and fragment) launched on top of activities. GATOR constructs a WTG in three stages. In the first stage, GATOR constructs the forward edges. A forward edge denotes the action of launching a new activity or triggering some default event (i.e., rotate, home, power, or menu) on the current activity. In the second stage, GATOR includes operations of "close window" in the WTG. Since we care only about whether one activity can launch another activity and the "close window" operation does not provide that information, we skip the second stage to speed up the WTG-construction process. In the third stage, GATOR includes edges triggered by the BACK button. For this study, we use only the WTG obtained from the first stage to derive the ATG for our further analysis.

By studying critical activities, we aim to answer the following three questions.

- **RQ1. Potential Coverage Boost**: On average, how many subsequent activities can a testing tool have a chance to explore by covering a critical activity?
- **RQ2. Critical Events**: What are common critical events, i.e., events needed to cover critical activities?
- **RQ3. Characteristics**: What are the characteristics of the critical activities?

RQ1. Potential Coverage Boost. The ATG possesses 428 nodes. Each represents a unique activity in the app and 128 out of those 428 activities are covered by our optimized approach described in Section III-B. After traversing the ATG, we locate 20 critical activities. The 20 critical activities have 4,732 immediate successors. 3,724 of the 4,732 immediate successors are not-covered activities, which includes 84 unique not-covered activities. Such finding suggests the importance of exploring the critical activities, because these 20 critical activities are the dominating entry points to these 84 not-covered activities.

RQ2. Critical Events. We also investigate why these 20 critical activities are not covered. We find that each of the critical activities has many covered predecessors. In fact, the 20 critical activities have 10,585 covered predecessor activities in total (529 on average). In the ATG, these covered predecessor activities have in total 19,531 edges for the 20 critical activities (i.e., 19,531 events to potentially trigger the critical activities). We investigate these events and find that 8,252 events are `onKey` events (i.e., UI events); 11,153 events are `onActivityResult` events (i.e., events leading back to the previous activities); 126 are `onNewIntent` events (i.e., inter-component communications). As the results show, most of the events leading from a covered activity to the critical activities are `onActivityResult` events. Such finding suggests that to explore critical activities, an exploration tool (e.g., Monkey) should improve its chances to go back to previous activities so that these activities will more likely start critical activities in methods such as the `onActivityResult` method.

RQ3. Characteristics. We also investigate the characteristics of these 20 critical activities. Consistent with findings in Section VI-A, many (14 out of 20) of the critical activities are related to "wallet" functionalities. The distribution of other critical activities' characteristics are the following: 3 activities are about the "mall" functionality; 3 activities are used as the entry activity for third-party apps (i.e., triggered through intent). As we can see, these critical activities are typically not used for WeChat's core functionalities (e.g., chatting, social network). This finding suggests that specific testing scenarios lacking in existing tools are needed to cover these critical activities. For example, during Monkey runs, the activities used for third-party app access are infeasible to be covered because by default Moneky always starts from the app under test rather than third-party apps.

VII. Related Work

Industry Testing Tools for Android. Monkey, as part of the Android development bundle, generates pseudo-random sequences of user and system events that can be used to stress-test a given Android app. Testers can also use monkeyrunner [4] to send UI commands to an Android device or emulator from outside of Android code. The monkeyrunner tool differs from Monkey: monkeyrunner controls an emulator or a device by sending commands from a workstation, whereas Monkey generate events directly on the device. Robotium [6] and Espresso [3] are both testing frameworks built on top of the Android testing framework to automate UI test cases for Android apps. Robotium can be used for testing both apps whose source code is available and apps where only the APK is available. Robolectric [5] is a framework that supports running Android tests out of an emulator or device. Different from running tests in a mocked environment, Robolectric makes the tests more effective for refactoring and allowing the tests to focus on the behavior of the app under test instead of the implementation of Android. Barista [1] is a testing framework that records and translates UI interactions between a user and app into UI test cases.

Automated Testing Techniques for Android. TEMA [17] is a model-based testing tool for testing Android apps. Models are created manually and known to be error-prone. Dynodroid [9], [14] applies concolic execution to generate and symbolically analyze feasible event sequences for Android apps. Dynodroid also includes an effective criterion for pruning away many redundant event sequences. However, even with the pruning, Dynodroid is only applicable to fairly short event sequences, because of the computing-intensive nature of symbolic analysis and explosion in the sheer number of event sequences being enumerated exhaustively. JPF [16] has been used to deduce the set of feasible event sequences on Android apps and represent them using context-free grammar (CFG). The deduced event sequences are then analyzed through symbolic execution. ORBIT [19] statically analyzes the source code of the app under test to understand which UI events are relevant for a specific activity and builds a model of the app under test by crawling it from a starting state. PUMA [13] is a framework that helps testers incorporate Monkey's basic exploration strategy into dynamic analysis on Android apps. It provides a finite-state-machine representation of the app under test for testers to implement different exploration strategies.

VIII. Conclusion

In this paper, we have presented two optimization techniques to improve the effectiveness and efficiency of our previous approach extended from Monkey. We have also presented manual categorization of not-covered activities and two automatic coverage-analysis techniques (i.e., substring hole analysis and ATG analysis) to provide insightful information about not-covered activities.

IX. Acknowledgments

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

References

[1] Barista: an Android framework for making automated tests. https://moquality.com/barista/.

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

[3] Espresso: a unit test framework for Android. https://google.github.io/android-testing-support-library/docs/espresso/.

[4] monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html.

[5] Robolectric: a unit test framework for Android. http://robolectric.org/.

[6] Robotium: a unit test framework for Android. https://github.com/robotiumtech/robotium.

[7] Y. Adler, N. Behar, O. Raz, O. Shehory, N. Steindler, S. Ur, and A. Zlotnick. Code coverage analysis in practice for large systems. In *Proc. ICSE*, pages 736–745, 2011.

[8] Y. Adler, E. Farchi, M. Klausner, D. Pelleg, O. Raz, M. Shochat, S. Ur, and A. Zlotnick. Advanced code coverage analysis using substring holes. In *Proc. ISSTA*, pages 37–46, 2009.

[9] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. FSE*, pages 599–609, 2012.

[10] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proc. OOPSLA*, pages 641–660, 2013.

[11] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proc. OOPSLA*, pages 623–640, 2013.

[12] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *Proc. ASE*, pages 429–440, 2015.

[13] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proc. MobiSys*, pages 204–217, 2014.

[14] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proc. ESEC/FSE*, pages 224–234, 2013.

[15] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *Proc. FSE*, pages 599–609, 2014.

[16] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. In *Proc. JPF*, 2012.

[17] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *Proc. ICST*, pages 377–386, 2011.

[18] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *Proc. ICSE*, pages 89–99, 2015.

[19] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proc. FASE*, pages 250–265, 2013.

[20] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie. Automated test input generation for Android: Are we really there yet in an industrial case? In *Proc. FSE, Industry Track*, pages 987–992, 2016.