Paladin: Automated Generation of Reproducible Test Cases for Android Apps

Yun Ma[1,2], Yangyang Huang[2], Ziniu Hu[2], Xusheng Xiao[3], Xuanzhe Liu[2*]
[1]Tsinghua University [2]Peking University [3]Case Western Reserve University
yunma@tsinghua.edu.cn,{huangyangyang,bull}@pku.edu.cn,xusheng.xiao@case.edu,xzl@pku.edu.cn

ABSTRACT

Automated-test-generation tools generate test cases to enable dynamic analysis of Android apps, such as functional testing. These tools build a GUI model to describe the app states during the app execution, and generate a script that performs actions on UI widgets to form a test case. However, when the test cases are re-executed, the apps under analysis often do not behave consistently. The major reasons for such *limited reproducibility* are due to (1) backend-service dependencies that cause non-determinism in app behaviors and (2) the severe fragmentation of Android platform (i.e., the alarming number of different Android OS versions in vendor-customized devices). To address these challenges, we design and implement Paladin, a novel system that generates *reproducible test cases* for Android apps. The key insight of Paladin is to provide a GUI model that leverages the structure of the GUI view tree to identify equivalent app states, since the structure can tolerate the changes on the UI contents for an app behavior performed in different test executions. Based on the model, Paladin can search the view tree to locate the desired UI widgets to trigger events and drive the app exploration to reach the desired app states, making the test cases reproducible. Evaluation results on real apps show that Paladin could reach a much higher reproduction ratio than the state-of-the-art tools when the generated test cases are re-executed across different device configurations. In addition, benefiting from the reproducible capability, Paladin is able to cover more app behaviors compared with the existing tools.

KEYWORDS

Automated test generation; reproducible; Android app

ACM Reference Format:
Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, Xuanzhe Liu. 2019. Paladin: Automated Generation of Reproducible Test Cases for Android Apps. In *The 20th International Workshop on Mobile Computing Systems and Applications (HotMobile '19), February 27-28, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3301293.3302363

1 INTRODUCTION

Recently, automated test generation has emerged as a promising approach to enabling various dynamic analyses of mobile apps, such as conducting functional testing, detecting security vulnerabilities, and checking compliance issues [7]. Researchers and practitioners have proposed many automated-test-generation (ATG) tools for Android apps [6] given the open-source nature and the high popularity of the Android platform. To generate a test case, these tools build a GUI model to represent the app states during the app execution, and generate a script that performs a sequence of interactions with the UI widgets and verifies outputs.

While the generated test cases can be used to explore the apps dynamically, the apps often do not behave consistently when the test cases are re-executed. That is, the test cases generated by the existing tools are limited in achieving ***reproducibility***. Reproducibility of test cases is crucial in enabling various analyses of the app behaviors to improve app quality. For example, reproducibility enables developers to detect bugs and locate the causes as well as to conduct regression testing after fixing the bugs. But non-reproducible test cases will make it very difficult to re-trigger the bugs and perform regression testing since some behaviors may not be performed in some executions. Another application of reproducible test cases is to execute the same test cases on different device configurations to detect compatibility issues.

In this paper, to investigate the reproducibility of the test cases generated by ATG tools, we first conduct a characteristics study on the existing ATG tools as well as the record-and-replay tools. Our study shows that the test cases generated by the state-of-the-art tools are limited in achieving reproducibility even if the test cases can be used as the recorded scripts for replaying behaviors.

The major reasons are in two folds. On one hand, the dependencies of the backend services introduce non-determinism in app behaviors and outcomes. For example, a news app may randomly prompt ads on the news reading page, leading to slight changes in the page layout. As a result, the test cases generated when the page does not have ads may not be re-executed when the ads are prompted. On the other hand, the open nature of Android has led to a large number of devices with various configurations, which is known as the fragmentation issue [10]. For example, different devices have different sizes and resolutions of the screen, and different Android versions may render the same page in different ways. As a result, test cases generated on one device configuration may behave differently when they are re-executed on other device configurations.

To address the issues, we present Paladin[1], a novel system to generate reproducible test cases for Android apps. Our key insight is that for a specific app behavior, while service dependencies and device differences may cause differences on the UI contents, the structure of the view tree remains almost the same. Based on this insight, Paladin encodes the complete structure of the view tree in the model rather than just considering the UI contents. As such,

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotMobile '19, February 27–28, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6273-3/19/02...$15.00
https://doi.org/10.1145/3301293.3302363

[1]Paladin is publicly available at https://github.com/pkuoslab/Paladin

Table 1: Characteristics study of automated-test-generation and record-and-replay tools.

Tools	State Equivalence	UI Widget Location
Automated-Test-Generation (ATG) Tools		
AppDoctor	GUI Content	Coordinates
Collider	Event Handler	N/A
CrashScope	GUI Content	Coordinates
JPF-droid	Method Invocation	Resource ID, Name, Label
PUMA	GUI Feature Vector	Coordinates
Droidbot	GUI Content	Coordinates
Stoat	String Encoding of View Tree	Widget Indexes
Record-and-Replay Tools		
RERAN	N/A	Coordinates
VELERA	Visual Perception	Coordinates
Mosaic	N/A	Coordinates, Scale
Barista	Widget Existence	Resource ID, XPath
ODBR	N/A	Coordinates
SPAG-C	Image Comparison	Coordinates
MonkeyRunner	N/A	Coordinates
Culebra	GUI Content	Resource ID, Text, Content Description

our model can precisely identify equivalent app states and find the desired UI widgets, which can tolerate the differences of the UI contents. Additionally, since reproducing some behaviors may be required to trigger certain follow-up behaviors, the model enables test generation to reach higher coverage of app behaviors. In summary, Paladin has the following advantages:

- **Automated Test Generation.** Given an app under analysis, Paladin automatically explores it to build a GUI model, and generates a test case for each app state.
- **Reproducible Executions.** The generated test cases can be re-executed to ensure consistent app behaviors across different device configurations.
- **Better Behavior Exposures.** Paladin is able to run on complex commercial apps and cover more behaviors compared with existing ATG tools.

2 MOTIVATION & RELATED WORK

The execution of an Android app is driven by phone events, such as button clicks and SMS arriving. Thus, a test case for app analysis is represented as a sequence of phone events, where each phone event triggers a transition of app states. In this paper, we consider only the UI events such as click and scroll, which are triggered explicitly by user interactions. System events such as message notifications and network connections are left for future work. Besides, we take into account the GUI states, which focus on the UI layouts and widgets that are perceivable by users. We define that a test case of an app is reproducible if the app could perform consistent behaviors to reach the same states when the test case is re-executed in different device configurations, including the original device configuration where the test case is generated and other device configurations. Here, we regard the device configuration as the device model and Android version.

Based on the above definitions, reproducibility can be achieved if all the events in a test case are triggered in an expected order and produce the desired app states. This brings two problems to be solved, state equivalence and widget location. On one hand, how to identify equivalent app states should be designed carefully. Considering only the UI contents is likely to cause state space explosion, and considering only the page classes (i.e., activities on Android) is not enough to reproduce the desired behaviors. More importantly, the changes caused by the backend-service dependencies and the Android fragmentation issue must be considered for driving the app to a desired UI state and interacting with the desired UI widgets such as buttons.

Before presenting our design, we first make a characteristic study to investigate the current support of generating reproducible test cases. Based on a recent survey on Android testing [9], we select 7 ATG tools of which the generated test cases are reported to be reproducible. We also select 8 record-and-replay tools from the same survey to study whether they can be integrated with the existing ATG tools to achieve reproducibility by using the generated test case as the recorded script. We focus on the state equivalence and the UI widget location adopted by these tools. Table 1 summarizes the analysis results.

2.1 State Equivalence

Most ATG tools consider two app states as equivalent ones if they have identical GUI contents. Obviously, such a strict definition would easily cause state explosion on today's complicated apps. For example, due to the non-deterministic behaviors caused by backend-service dependencies, the UI content varies slightly (e.g., pop-up ads on side bar) even when the same sequence of events are triggered, producing a daunting number of states for one page. Collider defines a state as a combination of registered event handlers and transitions as execution of event handlers. JPF-droid examines method invocations to verify states. Theoretically, method or handler invocations are more suitable than GUI changes to detect equivalent states for app exploration. But it requires instrumentation on the source code or byte-code level to emit events that indicate changes in the program state, which is difficult for complicated commercial apps. PUMA constructs a GUI feature vector and uses the cosine-similarity metric with a user-specified similarity threshold to identify equivalent states. Stoat encodes the string of the GUI view tree as a hash value to distinguish states. The problem is that the feature vector and the string encoding cannot easily adapt to different device configurations.

4 out of 8 record-and-replay tools do not specify state equivalence precisely. VELERA uses human visual perception to judge the test results, which is impractical for large-scale analysis. Culebra uses the GUI content to identify equivalent states, suffering from state explosion as discussed before. SPAG-C uses image comparison to identify equivalent states. However, image comparison is very susceptible to slight GUI changes such as different font styles and color settings.

2.2 UI Widget Location

There are 10 out of the total 15 examined tools that use the coordinates to locate widgets. However, considering only the absolute position of UI widgets is incapable of reusing test scripts on the devices with different screen resolutions. It is also error-prone for the GUI changes caused by different responses from backend services. Mosaic uses a series of scalings and normalizations to map coordinates between platforms. However, UI widgets do not simply scale linearly with screen dimensions. Some apps rearrange and even hide UI widgets based on the screen resolution. JPF-droid and Culebra take resource ID, label texts, and content descriptions into

consideration. But UI widgets often share these properties, making it difficult to precisely locate the desired widgets. Barista uses XPath selector to locate UI widgets since the GUI view tree can be mapped to an XML document. But widget classes that constitute the XPath tags may differ in different Android versions. Stoat locates UI widgets by object indexes, which are likely to change under different device configurations.

In summary, test cases generated by the state-of-the-art tools are limited in achieving reproducibility even if the generated test cases can be used as the recorded scripts for record-and-replay tools.

3 APPROACH

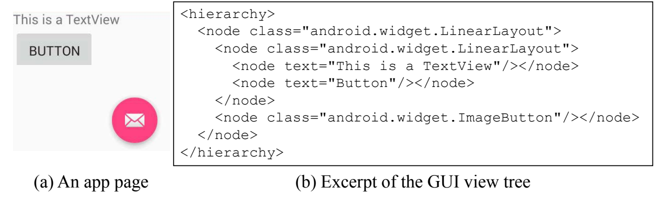
The key insight of Paladin is that the structure of the GUI view tree of a specific app page remains almost the same regardless of the changes of the contents and layouts when a test case is re-executed in different device configurations. Therefore, we use the complete structure of the GUI view tree to build a structure-preserving model for identifying equivalent app states and locating UI widgets precisely. We next describe the details of the fundamental design of equivalent state identification and UI widget location.

3.1 State Equivalence

In Android, all the UI widgets of an app page are organized in a GUI view tree, similar to the DOM tree of a web page. After the app page is rendered, the view tree can be retrieved via UI Automator [5], which is a tool provided by the Android SDK. Figure 1(a) shows an app page which contains a text, a button, and an image button. Figure 1(b) shows an excerpt of the retrieved view tree. We can see that the text and the button are organized in a linear layout, and the linear layout together with the image button are organized in another linear layout. Formally, a view tree is represented by a tuple $VT = < N, E >$. Here, N is the set of view nodes, of which some are directly exhibited on the page (the text node, button node, and image button node) and the others are used to organize the layout (the two linear layout nodes). $E \subset N \times N$ represents the parent-child relationship between nodes, where $e(n_1, n_2) \in E$ if n_1 is the parent node of n_2.

We propose to use the structure of the GUI view tree to identify equivalent app states. This design is inspired by the empirical fact that the view trees of GUIs produced by the same app behavior often share the similar structures, but different app behaviors typically result in different view trees. For example, the pages that show the details of different restaurants have the same structure, but the pages showing restaurant details and those showing a list of restaurant search results are obviously different in the structure of the view tree. Moreover, the rendered pages produced by a specific app behavior have similar view trees across different device configurations. Therefore, using the structure of the view tree to identify equivalent states can address the fragmentation issue.

In order to rapidly compare the view trees and identify the differences between two view trees, we encode each node of the view tree into a hash value and maintain a hash tree. The hash value of the root node can be used to distinguish app states. The hash function should meet two requirements. On one hand, the hash values should be different for view trees whose structures are different. On the other hand, given two view trees which have slightly



(a) An app page (b) Excerpt of the GUI view tree

Figure 1: An example of a view tree.

differences, it should be efficient to find the differences in the structure based on the hash values. To meet these two requirements, we design a recursive bottom-up algorithm to compute the hash value of each node as shown in Algorithm 3.1. The algorithm accepts a view node r as input. If r has children nodes (Line 5), we use the algorithm to calculate all the hash values of its children recursively (Lines 7-9). Then, the algorithm sorts r's children based on their hash values to ensure the consistency of the structure hash value (Line 10), since a view's children do not keep the same order every time. Later, we concatenate the children's hash as $r.hashString$, and return the hash value of $r.hashString$ (Lines 11-15). If r does not have children nodes, the result is only the hash value of r's view tag and resource ID (Line 18). Given the root node of the view tree, the algorithm returns a hash value, which can be used to identify equivalent states.

```
   Input: View r
   Output: Structure Hash h
1  function TreeHash(r)
2  if r.InstanceOfWebview() then
3  |   r.setChildren ← r.parseHTML()
4  end
5  if r.hasChildren() then
6  |   children ← r.getChildren()
7  |   foreach c ∈ children do
8  |   |   c.hash ← TreeHash(c)
9  |   end
10 |   children ← SortByHash(r.getChildren())
11 |   r.hashString ← ""
12 |   foreach c ∈ children do
13 |   |   r.hashString ← r.hashString + c.hash
14 |   end
15 |   return hash(r.hashString)
16 end
17 else
18 |   return hash(r.viewTag + r.resourceId)
19 end
```

Algorithm 3.1: Computing the hash value of a view node.

This encoding method can be extended to Web elements as well. Since there is a plenty of hybrid apps which use HTML in WebView to display GUIs, we also incorporate the Web elements inside WebView component into the view tree (Line 3) rather than treating the WebView as a leaf node. So our model is applicable for hybrid apps.

Due to the non-determinism in app behaviors, the GUI may change in certain parts when the same sequence of events are triggered, and a trivial GUI change may result in a totally different hash value. For example, when a test case that explores the news page is re-executed, a notification about the news update may appear on the top of the screen. Since it is only a trivial GUI change and does not influence the majority of the other functionality, we should tolerate this difference and consider it to reach the expected state. Therefore, in order to make the model adaptive to changes, we design a structural similarity criteria. Given a computed GUI

state triggered by an event, only when the similarity difference between the GUI's state and our stored state is above a threshold, we would regard it as a new state. Algorithm 3.2 shows how the similarity score is computed.

The algorithm accepts two view nodes and a threshold as inputs. If the two view nodes share the same hash value, their structures are the same, so the similarity equals 1 (Lines 2-4). Otherwise, there must be some differences between them. So we enumerate the children of these two nodes, and compute the similarities of the children nodes (Lines 9-17). To reduce the complexity, we will stop traversing if any pair of children nodes exceed the threshold (Lines 12-15). The similarity score is twice the number of nodes that are considered as the same divided by the total number of nodes including the two view nodes and all of their descendant nodes (Line 18).

```
   Input: View s, View t, threshold τ
   Output: Structural Similarity sim
1  function Similarity(s, t)
2  if s.hash = t.hash then
3  |   return 1
4  end
5  hits ← 0
6  if s.tag = t.tag then
7  |   hits ← hits + 1
8  end
9  foreach sc ∈ s.getChildren() do
10 |   foreach tc ∈ t.getChildren() do
11 |   |   tmp ← Similarity(sc, tc)
12 |   |   if tmp > τ then
13 |   |   |   hits ← hits + tmp * tc.count
14 |   |   |   break
15 |   |   end
16 |   end
17 end
18 return 2 * hits/(s.count + c.count)
```

Algorithm 3.2: Structural similarity of two view trees.

3.2 Locating UI Widget

To ensure the reproducibility of test cases, we also need to precisely locate the UI widgets to trigger the desired events during test-case execution. Under such a condition, when a generated test case is re-executed, each event can be triggered at the same UI widget. Based on the GUI model designed above, a UI widget is represented using the hash values from the root node to the node of the UI widget. We summarize 3 cases in Figure 2 to illustrate the problems of locating UI widgets when the view tree of the same page is changed. Consider an example view tree shown in Figure 2(a). Assume that the leaf node with the dashed border line is to be located. Figure 2(b) shows Case 1, where there is a change on the black node (e.g. android.view.View in Android 5.1 is replaced by android.view.ViewGroup in Android 6.0+ instead). Figure 2(c) shows Case 2, where a widget is appended as a new leaf node (e.g. ads pop up in the current page). Figure 2(d) shows Case 3, where a deletion of irrelevant leaf node (e.g. settings to not display some UI widgets). In all these three cases, the view tree is changed. As a result, the recorded hash values could not be used to locate the corresponding UI widgets.

We design a heuristic search algorithm to locate a UI widget. As shown in Algorithm 3.3, it accepts a widget of the current view tree, a widget of the recorded view tree, the hash list of the target widget to be matched, and the current index in the hash list as input. When the index is equal to the size of the hash list, the algorithm returns

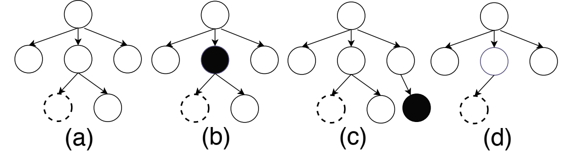


Figure 2: Cases of locating widgets.

(Line 3). Otherwise, it first checks whether the children of the current widget contains the next hash value. If yes, the algorithm will recursively go down to check the corresponding child widget (Line 9). If not, it would exclude the intersection of the children of current widget and the recorded widget to narrow down the search space, and try to recursively go down to check the unmatched child widgets (Line 12-18). For Case 1, since a substitution of a non-leaf node would not change the hash value of the node, we can still locate the recorded widget. However, for Case 2, the hash value of the root node changes after adding a new leaf node, which is equal to the hash of concatenation of all the children's hash value. The algorithm will detect that one child of the current node contains the next hash value (Line 7) and finally locate the widget. As for Case 3, all of the hash values change before reaching the target widget after deleting a leaf node. The algorithm will fail to check the next recorded hash value in Line 7 and jump to Line 12. Then, the algorithm will filter two children of current view that can be mapped to the recorded view and choose the middle child to explore, where it can precisely find the recorded widget.

```
   Input: Current Widget cv, Recorded Widget rv, Hash List L, Index i
   Output: Target Widget tw
1  function LocateWidget(cv, rv, L, i)
2  if i = L.size() then
3  |   return cv.hash = L[i]?cv : null
4  end
5  children ← cv.children()
6  rv′ ← rv.children().get(L[i + 1])
7  if children.has(L[i + 1]) then
8  |   cv′ ← children.get(L[i + 1])
9  |   widget ← LocateWidget(cv′, rv′, L, i + 1)
10 end
11 else
12 |   children′ ← rv.children()
13 |   foreach child ∈ (children − (children ∩ children′)) do
14 |   |   widget ← LocateWidget(child, rv′, L, i + 1)
15 |   |   if widget! = null then
16 |   |   |   break
17 |   |   end
18 |   end
19 end
20 return widget
```

Algorithm 3.3: Locating a specific UI widget.

4 IMPLEMENTATION

We implement Paladin based on the standard Android SDK without requiring any instrumentation on the Android system or the app. Therefore, Paladin can be launched on any Android device and can work for commercial apps. Paladin consists of three main components: explorer, test generator, and executor interface. Given an app under analysis, the explorer first explores the app behaviors to construct the GUI model. Based on the model, the test generator is used to generate a test case for each state in the model. Each test case could be re-executed to reproduce the corresponding app

state. The bottom layer of Paladin is the executor interface, which is responsible for connecting with the device to retrieve GUI information and execute commands. The details of each component are as follows.

Explorer exercises the app behaviors under a certain search strategy and builds the GUI model. Our current implementation adopts the most common search strategy, depth-first search. Nonetheless, Paladin can be easily integrated with other search strategies. For each state, we extract all the UI widgets that listen to events, and then systematically trigger the events of each widget. Next, we check whether the event brings the app to an equivalent state by comparing its structural hash value with all the other states in the model. If a new state is identified, we continue to apply the exploring algorithm on the new state. When the exploration on one state terminates, the app will backtrack to the previous state, continuing exploration.

Test Generator generates test cases for the states in the model. For each state, we try to find paths from the entry state of the model to the target state. This can be done by a standard breadth-first search. Starting from the target state, we enumerate every potential state path, and sorts them by their length. The shortest path is used to generate the test case.

Executor Interface communicates with the app execution environment to retrieve information and trigger events. We use UI Automator to retrieve the GUI view tree of app pages and execute ADB commands to trigger events. For web pages in WebView, we use Chrome Remote Debugging protocol to retrieve the DOM tree and trigger DOM events.

5 EVALUATION

In this section, we evaluate Paladin by answering two research questions: 1) How effective can Paladin ensure the app behaviors to be reproduced when the generated test cases are executed under different environments? 2) By encoding the view trees, could Paladin automatically generate test cases to cover more behaviors of real apps, especially complex commercial apps?

5.1 Reproducibility

According to our characteristics study in Section 2, the state-of-the-art ATG tools cannot generate reproducible test cases to run on different device configurations. As a result, to study the first research question, we compare Paladin with two record-and-replay tools, Monkeyrunner [4] which uses coordinate to locate widgets, and Culebra [2] which uses widget attributes to locate widgets.

We select five commercial apps from diverse categories but with the common feature of heavy dependencies on backend services so that the GUI contents and layouts are likely to change when the test cases are re-executed. Specifically, WeChat is one of the most popular instant messaging apps all over the world. QQreader is a reader app which often releases new novels on its home page. Picfolder is a tool app used to manage photos. EasyLoan is a financial app with frequently updated loan information. Missfresh is an electrical business app that regularly update recommended goods.

We run Paladin on each app for 15 minutes to explore app behaviors. Then we select one state from each app, representing a specific behavior, and generate a test case for the selected state. The selected state should be reached by performing at least 3 UI interactions. Then we manually record the same UI interactions with Monkeyrunner and Culebra, generating the corresponding test scripts.

In order to investigate the reproducibility of generated test cases under different environments, we re-execute each test case in four scenarios. 1) Base line: each test case is executed on the same device where it is generated. We use the Nexus 6 with Android 7.1 OS, which has a 5.96-inch screen size and 2560x1440 resolution, to generate the test cases. 2) Across device models: each test case is executed on a Samsung S4 with Android 7.1 OS, which has a 5.0-inch screen size and 1920x1080 resolution. 3) Across Android OSes: each test case is executed on another Nexus 6 but with Android 5.0 OS. 4) Time evolving: each test case is executed on the same device where it is generated but after one day from the time when the test case was generated.

We use the reproduction ratio to measure the reproducibility, which is computed as the ratio of the correctly executed UI interactions over the number of recorded UI interactions during recording. Figure 3 shows the comparison results.

Under the same environment (Figure 3(a)). Both Paladin and Monkeyrunner can successfully execute all the test cases in selected apps. Culebra can only successfully execute cases in 2 apps (Wechat and QuickLoan) and totally fails to execute the test case of QQreader because Culebra highly relies on widget id and text descriptions. For example, the test case for QQreader is to open a novel and configure a different font. The home page of QQreader displays the reading percentage of each novel and thus changes from 0% to 2% after the initial record, resulting Culebra fails to identify the entry.

Across device models (Figure 3(b)). Monkeyrunner fails to perform correctly across device models because it considers only the coordinates of widgets. Culebra performs better than Monkeyrunner, but is still severely interfered by its dependencies on the widget id and the text descriptions. Paladin almost successfully executes all the test cases.

Across Android versions (Figure 3(c)). Paladin and Monkeyrunner perform equally well, but Paladin works much better in WeChat than Monkeyrunner. Culebra fails to replay most of the cases. The main reason is the minor differences of view structures found in different Android OSes.

Time evolving (Figure 3(d)). Paladin successfully executes all test cases, showing strong adaptive capacity to slight GUI changes. Monkeyrunner fails in QuickLoan, because a popup advertisement has changed the location of one widget slightly. Culebra behaves the worst due to extended changes in GUI.

5.2 Coverage

To study the second research question, we compare Paladin with three ATG tools in terms of coverage on app behaviors. Two tools are chosen from the study made by Choudhary et al. [6] where we tried all the surveyed tools but only Monkey [1] and PUMA [7] can work at the current commercial apps. We also choose a very recent tool Stoat designed by Su et al. [8], which is reported to be more effective than the state-of-the-art techniques.

We select two sets of Android apps for evaluation. One is from the open-source apps used in the study of Choudhary et al. [6]. We filter the apps with only one activity for which the ATG tools can easily reach very high coverage. After filtering, 22 apps are remained. The other set is from the most popular apps on Wandoujia, which is

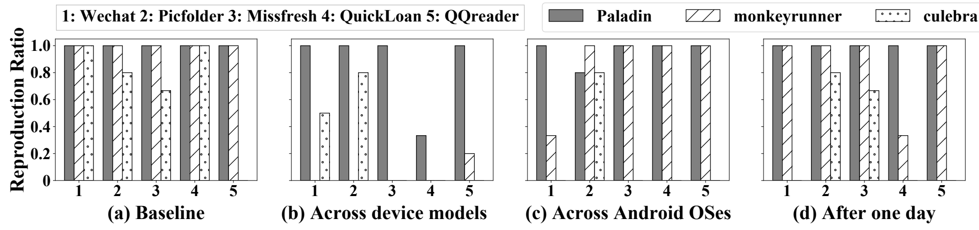


Figure 3: Comparison results of reproducibility.

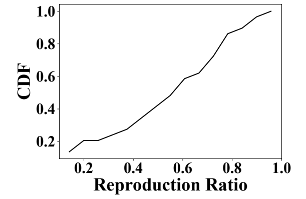


Figure 5: Reproduction ratio of 72 apps.

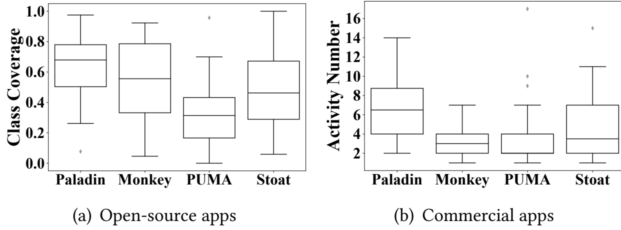


Figure 4: Comparison results of behaviors coverage.

a leading Android app marketplace in China. We tried 235 most popular apps to obtain 50 apps on which all the comparison tools can run. Actually, Paladin and Monkey can run all the 235 apps, but PUMA and Stoat can run only part of them. Such a result indicates the better compatibility of Paladin for complex commercial apps.

To quantify the behaviors coverage, we use the class coverage reported by Emma [3] for open-source apps. For commercial apps, we use the activity number counted from the logcat utility, which is a log tool provided by Android SDK. We deploy each tool on a dedicated Nexus 6 smartphone (4-core 2.7GHz CPU and 3GB LPDDR3 memory), and run all the four tools simultaneously on each app. We set one hour as the timeout for exploring an app.

Figure 4 shows the comparison results of behaviors coverage. For open-source apps, Figure 4(a) shows that Paladin reaches a higher coverage (68% in the median case) than the other ATG tools (56% of Monkey). Meanwhile, Paladin's structure-preserving model helps to eliminate the extreme deviations, resulting in a smaller standard error of the coverage distribution. Figure 4(b) shows that Paladin performs much better in commercial apps than the other tools. Paladin can explore 6.5 activities in the median case while the median number of explored activities of the other tools are below 4. Such a result can be attributed to the higher complexity of commercial apps and thus demonstrate the practical usage of Paladin. In fact, due to complex app behaviors, it is hard to judge whether such a coverage result is better enough, especially for commercial apps. We plan to manually explore the apps and compare the coverage in our future work.

Finally, we study how high percentage the generated test cases are reproducible on the same device. Since Monkey uses GUI content to distinguish states, one-hour exploration generates too many test cases which are impossible to enumerate. PUMA and Stoat has not provided interfaces to re-execute test cases so far. As a result, we study only Paladin. We execute the test cases generated for the 72 apps, and manually check whether the app reaches the same state as the one when the test case is generated. Assume that a GUI model consisting of n states is constructed by exploring an app, and the generated test cases can reproduce m states. Then the reproduction ratio is calculated by $\frac{m}{n}$. Figure 5 shows the distribution of the reproduction ratio among the 72 apps. The median reproduction ratio is 56.9%, meaning that 56.9% of test cases generated for an app can be re-executed to reproduce the same app behaviors.

6 CONCLUSION

This paper makes a first-step effort to automatically generate reproducible test cases for Android apps. We design and implement Paladin to achieve the goals of automated test generation, reproducible executions, and better behavior exposures. The key design of Paladin is to leverage the complete structure of the view tree to identify equivalent app states and locate UI widgets. Compared with the state-of-the-art tools, test cases generated by Paladin can be executed across different device configurations. In future work, since mobile devices are now supporting more pervasive tasks [11] leading to more origins of non-determinism in app behaviors, we plan to comprehensively study the non-deterministic app behaviors and enable the analysis of them.

ACKNOWLEDGMENTS

We thank our shepherd Alastair Beresford and anonymous reviewers for their feedback and suggestions. This work was supported by the National Key R&D Program under the grant number 2018YFB1004800, the National Natural Science Foundation of China under grant numbers 61725201, 61528201, 61529201, the Beijing Municipal Science and Technology Project under the grant number Z171100005117002, and China Postdoctoral Science Foundation.

REFERENCES

[1] 2018. Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.
[2] 2018. Culebra. https://github.com/dtmilano/AndroidViewClient/wiki/culebra.
[3] 2018. EMMA code coverage. https://sourceforge.net/projects/emma/.
[4] 2018. Monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html.
[5] 2018. UI Automator. https://developer.android.com/training/testing/ui-automator.
[6] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proc. of ASE 2015*. 429–440.
[7] 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 *Proc. of MobiSys 2014*. 204–217.
[8] 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 *Proc. of ESEC/FSE 2017*. 245–256.
[9] Mario Linares Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *Proc. of ICSME 2017*. 399–410.
[10] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps. In *Proc. of ASE 2016*. 226–237.
[11] Zuwei Yin, Chenshu Wu, Zheng Yang, and Yunhao Liu. 2017. Peer-to-Peer Indoor Navigation Using Smartphones. *IEEE Journal on Selected Areas in Communications* 35, 5 (2017), 1141–1153.