Understanding the Reproducibility Issues of Monkey for GUI Testing

Huiyu Liu[1], Qichao Kong[1], Jue Wang[2] (✉), Ting Su[1], and Haiying Sun[1] (✉)

[1] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
hysun@sei.ecnu.edu.cn
[2] Nanjing University, China

Abstract. Automated GUI testing is an essential activity in developing Android apps. Monkey is a widely used representative automated input generation (AIG) tool to efficiently and effectively detect crash bugs in Android apps. However, it faces challenges in reproducing the crash bugs it detects. To deeply understand the symptoms and root causes of these challenges, we conducted a comprehensive study on the reproducibility issues of Monkey with Android apps. We focused on Monkey's capability to reproduce crash bugs using its built-in replay functionality and explored the root causes of its failures. Specifically, we selected six popular open-source apps and conducted automated instrumentation on them to monitor the invocations of event handlers within the apps. Subsequently, we performed GUI testing with Monkey on these instrumented apps for 6,000 test cases and collected 56 unique crash bugs. For each bug, we replayed it 200 times using Monkey's replay function and calculated the success rate. Through manual analysis of screen recording files, log files of event handlers, and the source code of the apps, we pinpointed five root causes contributing to Monkey's reproducibility issues: Injection Failure, Event Ambiguity, Data Loading, Widget Loading, and Dynamic Content. Our research showed that only 36.6% of the replays successfully reproduced the crash bugs, shedding light on Monkey's limitations in consistently reproducing detected crash bugs. Additionally, we delved deep into the unsuccessfully reproduced replays to discern the root causes behind the reproducibility issues and offered insights for developing future AIG tools.

Keywords: Reproducibility, Empirical Study, Android GUI Testing

1 Introduction

The Android apps have become increasingly widespread [2]. In Android app development, GUI (Graphical User Interface) testing is crucial for ensuring the stability and reliability of Android apps. It aims to mitigate the risk of software failures, data breaches, and other potentially expensive issues. To support efficient and robust GUI testing, numerous AIG (Automated Input Generation)

tools have been developed [1,6,16,17,18,20,29,31,32,33]. By sending GUI events automatically to the app under test and monitoring its performance, these AIG tools effectively detect and report crash bugs in Android apps.

Despite the success of AIG tools in detecting crash bugs, the primary challenge in addressing these bugs is reproducing them. Most existing AIG tools lack replay functionality. As a result, testers must manually attempt to reproduce any reported crash bugs, which can be both challenging and labor-intensive. Even for AIG tools that offer replay functionalities (e.g., Monkey [23], Ape [11], DroidBot [16]), they cannot guarantee reliable reproduction due to varying runtime contexts [30], which brings challenges for testers to diagnose and fix these crash bugs. Flakiness in Android testing refers to the inconsistent results of automated tests. Most of the existing studies about flakiness in Android testing primarily focus on flaky tests in UI testing frameworks like Espresso. A few works have also studied the reproducibility issues associated with AIG tools [28,30]. However, these studies reproduce the buggy trace only a few times, which might not be comprehensive or systematic. We believe that a deep understanding of the reproducibility issues of AIG tools is crucial. Such insight can pinpoint the limitations of these tools and suggest directions for improvement.

Therefore, we conducted an in-depth study of the reproducibility issues of AIG tools. Specifically, we focus on Monkey [23], a widely adopted AIG tool for Android apps in industry [26,34]. Monkey provides the capability to simulate user interactions by generating random events such as clicks, touches, or gestures, as well as some system-level events. Moreover, it has a built-in functionality to replay the event sequences it generated. Google has officially integrated Monkey into the Android SDK (Software Development Kit) [24], highlighting the representativeness and significance of Monkey for Android GUI testing. To get insights on the reproducibility of Monkey, in this work, we primarily focus on the following two research questions:

- **RQ1**: How reliably can Monkey reproduce the crash bugs it detects?
- **RQ2**: What are the root causes of the reproducibility issues of Monkey?

To answer the two research questions, we selected six real-world, open-source Android apps, and conducted GUI testing on them with Monkey. When a crash bug was detected, we replayed it with Monkey's built-in functionality. Specifically, to get further information about the execution status of each replay, we implemented an instrumentation tool in the form of a Gradle Plugin using a Java bytecode manipulation and analysis framework - ASM [5]. Applying the plugin to our subject apps can help us observe how the app reacts to input events from the event handler level. We recorded the execution information of each replay, including the event sequence, screen recording, and the event handler invocations recorded by our instrumentation. Finally, we conducted a manual analysis of the unsuccessfully-reproduced replays and identified the root causes.

According to our experimental results, only 36.6% of the replays can successfully reproduce the crash bugs detected by Monkey on average. We categorized the root causes of the reproducibility issues into five types, namely *Injection Failure*, *Event Ambiguity*, *Data Loading*, *Widget Loading* and *Dynamic Content*.

The most prevalent cause, *Injection Failure*, stems from MONKEY's event generation mechanism, accounting for 54.4% of the 7,100 unsuccessful replays. *Event Ambiguity*, *Data Loading*, *Widget Loading*, and *Dynamic Content* account for 20.9%, 15.1%, 9.2%, and 0.4%, respectively. Our study reveals the limitation of MONKEY in reliably reproducing the crash bugs it detects, which can also be generalized to AIG tools implemented based on the principles of MONKEY. For researchers, understanding these limitations can guide the enhancement of existing AIG tools and the future development of the coming AIG tools. For developers, understanding these limitations can help to interpret the bug reports and assess whether the AIG tool's inputs truly represent the user interactions that led to the crash. It also allows developers to distinguish between genuine bugs and false positives (reporting a bug that doesn't exist) or false negatives (not detecting an actual bug) caused by the tool's inaccuracies.

Overall, the main contributions of this paper can be summarized as follows:

- To our knowledge, we conducted the first systematic and in-depth study to investigate MONKEY's reproducibility issues.
- To study the reproducibility issues of MONKEY, we implemented an instrumentation tool that can help observe the runtime behavior of the apps from the event handler level.
- We identified five root causes of MONKEY's reproducibility issues from different perspectives (e.g., MONKEY's self unreliability, app's feature), and some of these root causes (e.g., *Injection Failure*, *Event Ambiguity*) were not identified previously.
- We have publicly released our tool and dataset at *https://github.com/InstrumentDroid/InstrumentDroid*.

2 Empirical Study Methodology

This section presents our methodology for investigating the reproducibility rate of crash bugs detected by MONKEY and analyzing the root causes of the reproducibility issues of MONKEY. Fig.1 shows the methodology of our study.

2.1 Notations and Definitions

An Android app is a GUI-based event-driven program P. A crash bug r can be described as a type of fault in P that triggers a runtime exception resulting in an app crash or hang. A MONKEY event e is an event generated by MONKEY to simulate various user interactions, such as touch events, key events, and system events. Note that MONKEY is coordinate-sensitive rather than widget-sensitive, so not event MONKEY event interacts with UI elements. We denote these MONKEY events as blank events. When finished a MONKEY testing, P responds to a sequence of MONKEY events $E = [e_1, e_2, ..., e_n]$ and yields an execution trace $\tau = \langle E, I \rangle$, where E denotes the MONKEY event sequence and I denotes the event handler invocation sequence.

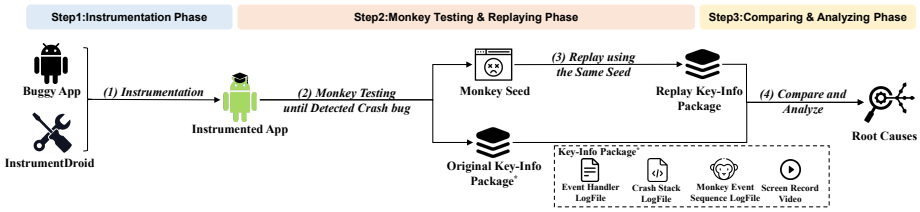

Fig. 1: Overview of our methodology including three major phases

2.2 Experimental Method

Step 1: Instrumentation Phase. To monitor the real-time execution sequence of GUI events during Monkey testing, we crafted an instrumentation tool, InstrumentDroid, which was capable of capturing the invocations of event handlers in the app. Event handlers are responsible for responding to the events generated by UI widgets when interacting with them. Therefore, InstrumentDroid hook into the event handlers corresponding to the UI widgets to get the GUI event sequences. Specifically, we first used a bytecode manipulation framework ASM to collect all the event handlers as an *Event Handler List*. Next, we developed the Gradle plugin InstrumentDroid. When applying InstrumentDroid to the apps, it can automatically scan all the event handlers in the app's source code and insert a piece of code into these event handlers to get runtime information about these event handlers. In this way, when users interact with a specific UI widget, its corresponding event handler is invoked, then the log information of this event handler is output to a designated file so that we can know which UI widget users are interacting with. Fig.2 shows the workflow of instrumentation.

Step 2: Monkey Testing & Replaying Phase. In this phase, we aim to conduct automated random GUI testing with Monkey on the instrumented apps until we get a series of unique crash bugs, and replay them with Monkey's built-in functionality.

Key-Info Package. To quantify app execution status, we recorded data termed *Key-Info Package*. A *Key-Info Package* consists of four components: *Event Handler LogFile*, *Crash Stack LogFile*, *Monkey Event Sequence LogFile* and *Screen Record Video*. The *Event Handler LogFile* is the output from InstrumentDroid, which documents the invocation state of event handlers. When an event handler is triggered, the *Event Handler LogFile* logs its invocation time, the related UI widget information, and its fully qualified name. For example, in Fig.3, lines 1-3 show the logs for *onOptionsItemSelected* invocation. Line 1 indicates the invocation time of *onOptionsItemSelected*. Line 2 indicates its UI widget details (with the resourceId of this Option Menu as '2131296485' and the Menu Option labeled 'History'). Line 3 gives its fully qualified name. The *Crash Stack LogFile* and *Monkey Event Sequence LogFile* are both outputs from Monkey. *Crash Stack LogFile* details crash events, including the crash location,

exception information, etc. *Monkey Event Sequence LogFile* records events and actions performed by MONKEY, including action types, coordinates, etc. Finally, *Screen Record Video* captures the real-time Android device display during the MONKEY test, showcasing all visual actions during the MONKEY testing process. It is obtained by the built-in screen recording functionality *screenrecord* of Android ADB.

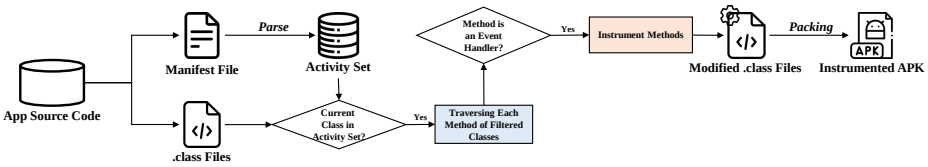

Fig. 2: Workflow of our instrumentation approach

Monkey Testing. We aim to use MONKEY for random GUI testing to collect unique crash bugs and their corresponding *Key-Info Package*. For each app, we set up 1,000 test cases, and each test case is a single MONKEY test. We customize the command-line parameters for each test, including throttle and event count. For off-line apps, half of the test cases have a throttle of 200ms and the other half 500ms. On-line apps have throttles set at 500ms and 1000ms, based on empirical findings from other automated testing studies [4,22]. We set an event count limit of 5,000 MONKEY events per test. If a crash happens within this limit, MONKEY stops and outputs the test's *Key-Info Package*. We use a shell script to manage the processes, from starting the emulator to extracting the *Key-Info Package*. Finally, we can get the execution traces and *Key-Info Package*s of these crash bugs. We denote the execution trace of a crash bug as τ_O, representing the original execution trace.

```
1 18:34:50.364/
2 131296485/History
3 com/amaze/filemanager/activities/MainActivity/onOptionsItemSelected
```

Fig. 3: Example of the output log of INSTRUMENTDROID

Replaying. A key feature of MONKEY is its capability to generate identical pseudo-random event sequences using a *Seed*. Leveraging this, we replay each τ_O 200 times with consistent seed and throttle settings, collecting the replay execution traces and their *Key-Info Package*s. The replay execution trace is denoted as τ_R.

Step 3: Comparing & Analyzing Phase. In this stage, our goals are (1) comparing each pair of τ_O and τ_R, determining how many replays successfully reproduced the crash bugs, and computing the reproducibility rate of each crash

bug, and (2) for the failed replays, analyzing the possible reasons for the reproducibility issues of these replays.

To achieve the first goal, given $\tau_O = \langle E_O, I_O \rangle$ and $\tau_R = \langle E_R, I_R \rangle$ with $E_O = [e_1, e_2, ..., e_m]$ and $E_R = [e_1, e_2, ..., e_n]$, we use two metrics to evaluate whether τ_R is successfully reproduced τ_O: (1) the index of MONKEY event causing the crash, and (2) exception information. The crash-causing event index refers to the last MONKEY event's index when the crash bug occurs. Specifically, if $m = n$, τ_O and τ_R crashed at the same event index. Exception information refers to the runtime exception when the crash occurs. Specifically, if τ_O and τ_R have the same exception type (e.g., *NullPointerException*) and description, they triggered the same runtime exception. If τ_R matches the event index and runtime exception of τ_O, we denote that τ_R successfully reproduced τ_O. Finally, for each crash bug, we calculated the percentage of successfully-reproduced replays out of all 200 replays as the reproducibility rate of this crash bug.

To achieve the second goal, we analyzed the *Event Handler LogFile* and the *Screen Record Video* of each pair of τ_O and τ_R. First, we preprocessed the *Event Handler LogFile*, abstracting each event handler's invocation as a GUI event, forming a GUI event sequence. Next, we compared the GUI event sequences of τ_O and τ_R, pinpointing differences. For the first divergent GUI event, we used the event execution times from the *Event Handler LogFile* to locate positions in the *Screen Record Video*. We compared frames around the divergence in both videos and manually determined the discrepancy's root causes. Specifically, for all the unsuccessfully-reproduced replays, two co-authors independently analyzed the root causes based on their knowledge and understanding. Then, the co-authors conducted cross-validation and discussion, reaching a consensus on the discrepancies. When they could not reach a consensus, the other three co-authors participated and helped make the final decision together. Manual analysis of all the unsuccessfully-reproduced replays was time-consuming, spanning about three months to complete. This analysis extended beyond *Key-Info Package*'s information to include specific bug lines in the source code. This makes the results more accurate and convincing.

2.3 Experimental Setup

Selecting Subject Apps. We selected six Android apps as the test subjects of our study. We focus on the open-source Android apps from GitHub so that we can monitor their execution to identify and analyze the root causes of the reproducibility issues. Our app selection was based on three criteria: popularity as indicated by GitHub stars, diversity of app categories for experimental validity, and feature variety. Specifically, we included both on-line (internet-required) and off-line apps, covering a wide range of app categories to enhance the validity of our experiment. We sourced all apps from Themis [30] and the app subjects of the recent empirical study conducted by Xiong *et al.* [35], because both of these datasets are recent studies of real-world bugs, meaning they are suitable for testing. From the intersection of these datasets, we initially picked three apps

including two off-line and one on-line. Since MONKEY lacks auto-login capabilities, our subsequent selections were non-login apps, ensuring diverse category representation. Ultimately, we selected four off-line and two on-line apps. The specifics of these six apps are detailed in Table 1.

Table 1: Six popular open-source and representative Android apps used in our study (K=1,000), '#Stars' indicates the number of GitHub Stars, '#LOC' indicates the number of lines of app source code.

App Name	App Category	#Stars	#LOC	Type
AmazeFileManager	File Manager	4.6K	94,768	Off-Line
AnkiDroid	Flashcard Learning	6.5K	218,558	Off-Line
ActivityDiary	Personal Diary	68	2,011	Off-Line
Sunflower	Gallery App	16.9K	1,687	Off-Line
AntennaPod	Podcast Manager	5K	90,427	On-Line
NewPipe	Video Player	24.2K	94,245	On-Line

Execution Environment. We conducted experiments on a physical machine with 128 GB RAM and a 64-cores AMD 3995WX CPU, running a 64-bit Ubuntu 20.04 operating system. To run the apps, we used a set of Android x86 emulators, where each emulator was configured with 4 CPU cores, 2 GB of RAM, 1 GB of SD card, and the version of Android OS 8.0 (API level 26). For each test case, the Android emulator is initialized to a fresh state at the beginning to provide a clean testing environment.

3 Experimental Results Analysis

During automated GUI testing, we ran 6,000 test cases across the six apps, collecting 56 unique crash bugs. After replaying each crash-triggering test case 200 times, we obtained 11,200 replays. Of these, 4,100 were successfully-reproduced replays, while 7,100 were not. In this section, RQ1 studied MONKEY's reproducibility rates, while RQ2 explores the root causes of its reproducibility issues.

3.1 RQ1: REPRODUCIBILITY RATE

Through a systematic analysis of 11,200 replays across six apps (four offline and two online), only 36.6% successfully reproduced the crash bug, indicating MONKEY's limitation in reliably reproducing the crash bugs it detected. Table 2 details the reproducibility rates for the 56 identified bugs, segmented into four categories: (1) "Same eid and Same Crash" where tau_R and tau_O have matching MONKEY event indexes leading to the crash and identical exception information, indicating successfully-reproduced replays; (2) "Different eid and Same Crash" where tau_R crash at differing event indexes but share tau_O's runtime exception; (3) "Different eid and Different Crash" where tau_R have a distinct event index

Table 2: List of the reproducibility rate of MONKEY on the six subject apps. '#Activities' indicates the number of activities in this app.

App Name	#Activities	Type	Throttle	Crash Bug Id	Same Eid Same Crash	Different Eid Same Crash	Different Eid Different Crash	No Crash
AmazeFileManager	10	Off-Line	200ms	Crash#1	178	4	0	18
				Crash#2	0	0	107	93
				Crash#3	5	1	1	193
				Crash#4	43	118	29	10
				Crash#5	163	0	6	31
			500ms	Crash#1	143	45	10	2
				Crash#2	161	14	21	4
				Crash#3	8	0	10	182
				Crash#4	1	1	12	186
				Crash#5	11	0	0	189
AnkiDroid	21	Off-Line	200ms	Crash#1	0	1	173	26
				Crash#2	96	1	1	102
				Crash#3	39	0	133	28
				Crash#4	0	0	0	200
				Crash#5	7	0	12	181
				Crash#6	0	0	194	6
			500ms	Crash#1	39	1	3	157
				Crash#2	0	64	134	2
				Crash#3	108	4	38	50
				Crash#4	98	0	90	12
				Crash#5	110	0	0	90
				Crash#6	91	0	71	38
				Crash#7	59	0	55	86
Sunflower	1	Off-Line	200ms	Crash#1	20	0	0	180
				Crash#2	177	23	0	0
				Crash#3	60	6	2	132
				Crash#4	121	15	1	63
				Crash#5	95	11	1	93
			500ms	Crash#1	0	0	0	200
				Crash#2	200	0	0	0
				Crash#3	196	0	0	4
				Crash#4	132	0	0	68
				Crash#5	134	0	0	66
ActivityDiary	7	Off-Line	200ms	Crash#1	1	0	0	199
				Crash#2	97	3	47	53
				Crash#3	89	0	28	83
				Crash#4	114	0	2	84
			500ms	Crash#1	45	0	0	155
				Crash#2	65	0	0	135
				Crash#3	0	0	5	195
				Crash#4	0	0	2	198
				Crash#5	180	0	13	7
AntennaPod	10	On-Line	500ms	Crash#1	200	0	0	0
				Crash#2	19	0	181	0
				Crash#3	46	0	0	154
				Crash#4	136	0	44	20
				Crash#5	0	0	21	179
				Crash#6	0	0	9	191
				Crash#7	18	0	4	178
			1000ms	Crash#1	200	0	0	0
				Crash#2	1	0	105	94
				Crash#3	18	0	0	182
				Crash#4	189	1	0	10
NewPipe	14	On-Line	500ms	Crash#1	174	0	3	23
			1000ms	Crash#1	3	0	53	144
				Crash#2	10	36	30	124
#Total					4100	349	1651	5100
#Proportion					36.6%	3.1%	14.7%	45.5%

and trigger a different bug; and (4) "No Crash" where no crash occurs within the 5,000 MONKEY events span.

Based on the experimental results, we have two interesting findings. First, the reproducibility rate of MONKEY is not significantly correlated with the runtime exception type. We conducted a statistical one-way ANOVA [9] of variance on the reproducibility rates of different exceptions using SPSS [13]. We first categorized the runtime exceptions of the 56 crash bugs and excluded exception types with sample sizes less than 3 due to their potential random occurrences that could affect experimental results. In the end, we obtained eight groups of exception types, including *NullPointerException*, *ClassCastException*, etc. Next, we employed the homogeneity of variance test and set the null hypothesis (H0) as follows: the reproducibility rates of crash bugs with different exception types are equal, indicating that reproducibility rates are independent of exception types. The results of the variance analysis showed a significance level (P-value) of 0.412, which is greater than 0.05. This implies that there is no significant correlation between the reproducibility rate and the exception types of crash bugs.

Second, the reproducibility rate of MONKEY is not significantly correlated with the app's complexity. Generally, apps with more complex features usually have more components in their activities, and different actions on these components may correspond to different functionalities, increasing the likelihood of executing error events. Apps with simpler features typically have simpler activity designs with more blank pages, resulting in a relatively higher probability of executing blank events. We conducted a similar ANOVA analysis between app features and the reproducibility rates, yielding a P-value of 0.441. This indicates that there is no significant correlation between them.

3.2 RQ2: ROOT CAUSE

By manually analyzing 7,100 unsuccessfully-reproduced replays' *Key-Info Packages*, we finally identified five root causes of the reproducibility issues in MONKEY: Injection Failure, Event Ambiguity, Data Loading, Widget Loading and Dynamic content. Table 3 shows the detailed proportion.

Injection Failure "Injection failure" describes situations where MONKEY experiences issues while inserting events into the *InputManager*, causing the event to not be added. Ideally, with the same *Seed*, MONKEY should generate consistent event sequences. However, our experiments revealed occasional event execution failures by the Android framework due to injection issues, denoted in the *Monkey Event Sequence LogFile*by "//Injection Failed". This results in inconsistencies between the original and replay execution traces, contributing to MONKEY's reproducibility challenges. In our study, 3,864 of the 7,100 problematic replays (or 54.4%) suffered from injection failure.

To understand the reasons behind injection failure, we conducted an in-depth analysis of MONKEY's source code. MONKEY has two types of event: KeyEvent, which corresponds to a physical button, and MotionEvent, such as click and long

press. For KeyEvent, if the action code of the KeyEvent is not a valid key action such as *UP* and *DOWN*, it will fail to be injected into the InputManager. For MotionEvent, if the pointer count (the multitouch number on the screen) of the MotionEvent is less than 1 or greater than 16, or any individual pointer ID in the MotionEvent is less than 0 or greater than the maximum allowed pointer ID, the MotionEvent will fail to be injected to the InputManager. A common case of Injection Failure is that the pointer count equals 0 when injecting MotionEvents due to the rapid event execution speed of Monkey.

Finding 1: *Injection Failure* affects 54.4% of the reproducibility issues, which is the most common root causes.

Event Ambiguity When recognizing actions, Android framework typically utilizes algorithms and rules to determine the type of action based on properties like pointer speed, distance, direction, and so on. Event ambiguity refers to the situation where the Android framework identifies the same Monkey event as different UI events, leading to a disparity between the original execution trace and the replay execution trace. This discrepancy contributes to the reproducibility issues of Monkey. In our experiment, 1483 out of 7100 replays with reproducibility issues (accounting for 20.9%) were attributed to event ambiguity.

For example, in the case of *Anki-Android*, the component *deck* has registered two different event handlers, which is shown in Fig.4. When clicking on a certain deck, *onClick* will be executed, while when long-clicking this deck, *onLongClick* will be executed. During GUI testing with Monkey, for the same Monkey event, the Android framework identified it in the original execution trace as a click event, but in the replay execution trace as a long-click event. Then this discrepancy led to the reproducibility issues. Fig.5 shows the real scenario of this example.

Finding 2: *Event Ambiguity* affects 20.9% of the reproducibility issues, which is an important factor affecting the reproducibility rate.

Data Loading Data loading refers to the situation where Monkey interacted with a partially loaded page or component, resulting in an empty event execution. Specifically, when switching to a new page, the app needs a period to fully load the content, and there will be a loading icon or some skeleton images on the page usually. Because Monkey is not widget-sensitive but coordinate-sensitive, when Monkey generates a click event, it may hit an area where the data is not yet available. That will possibly miss a pivot event. In our experiments, 1071 out of 7100 replays with reproducibility issues (accounting for 15.1%) are related to data loading. The reproducibility issues in Monkey caused by data loading can be fundamentally categorized into two types: database loading and network loading. Database loading refers to the situation where loading a new page or component requires retrieving information from a local or remote database. Such query operations typically take some time to complete. Network loading refers

```
holder.rl.setOnClickListener(
  new View.OnClickListener() {
     @Override
     public void onClick(View v) {
       //Handle events
     }
});
```

(a)

```
holder.rl.setOnLongClickListener(
  new View.OnLongClickListener() {
      @Override
      public boolean onLongClick(View p1) {
          //Handle events
      }
});
```

(b)

Fig. 4: Source Code of Event Ambiguity

to the situation where loading a new page or component requires some time to retrieve information from a remote server or certain APIs. This can lead to failures in reproducing actions accurately due to variations in network speed or connectivity, causing discrepancies between the original and replayed events. For instance, in *AmazeFileManager*, showcased in Fig.5(c) and (d), there's a noticeable difference in the file list display attributable to database loading speed. In Fig.5(c), while the file list was still populating, MONKEY clicked an empty space, maintaining the app's state. Conversely, in Fig.5(d), the app had efficiently fetched and displayed file data, leading MONKEY to click on the 'gr' folder and transition to a new page. Such inconsistencies, stemming from varied database loading rates, amplify MONKEY's reproducibility challenges.

Widget Loading Widget loading refers to the situation where the widget with animated effects is clicked before all the menu options have been fully displayed, leading to clicking the incorrect menu option during the replay process. In our experiments, 653 out of 7100 replays with reproducibility issues (accounting for 9.2%) are related to widget loading.

For example, in the case of *AmazeFileManager* illustrated in Fig.5(e) and (f), in the original execution trace, the click event landed on the *'GridView'* option within the Options Menu. However, in the replay execution trace, the dropdown speed of the Options Menu was slower, causing MONKEY not to click on any option within the Options Menu. As a result, the original execution trace and replay execution trace ended up on different pages.

Specifically, for further investigating widget loading, we manually analyzed the *Screen Record Video* and found that specific widgets like Drawer Menu and Options Menu typically require 60ms to 150ms (according to our video frame-by-frame statistics) to load completely to respond accurately to click events. In one particular experiment, we observed that when setting the throttle of MONKEY to 200ms, if a MONKEY event e_i triggered a pop-up of Options Menu, e_{i+1} had a 63% probability of being unable to select the well-prepared menu options. In addition, in a broader analysis encompassing four offline apps, when setting the throttle of MONKEY to 200ms, 58% of the pivot GUI events were missed or affected due to clicking on partially loaded widgets. This highlights the significance of timing and synchronization between click events and the loading of interactive widgets.

Table 3: List of the root causes of the reproducibility issues of MONKEY that we identified. A '-' in the 'Dynamic Content' column indicates that the off-line apps do not have this situation.

App Name	Type	Throttle	Injection Failure	Event Ambiguity	Data Loading	Widget Loading	Dynamic Content	Exception Type Of Crash Bug
AmazeFileManager	Off-Line	200ms	20	0	2	0	-	NullPointerException
			79	68	48	5	-	ClassCastException
			161	13	13	8	-	ClassCastException
			123	24	9	1	-	StringIndexOutOfBoundsException
			22	5	8	2	-	IllegalArgumentException
		500ms	28	26	0	3	-	NullPointerException
			28	6	4	1	-	StringIndexOutOfBoundsException
			151	36	1	4	-	ClassCastException
			175	12	11	1	-	StringIndexOutOfBoundsException
			175	3	10	1	-	NullPointerException
AnkiDroid	Off-Line	200ms	68	9	123	0	-	NullPointerException
			69	34	1	0	-	NullPointerException
			38	95	28	0	-	NullPointerException
			102	90	4	4	-	NullPointerException
			182	8	2	1	-	RuntimeException
			39	158	2	1	-	ArrayIndexOutOfBoundsException
		500ms	126	30	5	0	-	NullPointerException
			4	193	3	0	-	NullPointerException
			51	41	0	0	-	ActivityNotFoundException
			70	4	28	0	-	ArrayIndexOutOfBoundsException
			53	13	24	0	-	NullPointerException
			96	13	0	0	-	FileOutputStreamError
			117	17	7	0	-	IllegalArgumentException
Sunflower	Off-Line	200ms	77	8	0	95	-	IllegalArgumentException
			19	0	0	4	-	IllegalArgumentException
			97	1	0	42	-	IllegalArgumentException
			47	2	0	30	-	IllegalArgumentException
			65	3	0	37	-	IllegalArgumentException
		500ms	0	20	0	180	-	IllegalArgumentException
			0	0	0	0	-	IllegalArgumentException
			0	0	0	4	-	IllegalArgumentException
			0	7	0	61	-	IllegalArgumentException
			0	46	0	20	-	IllegalArgumentException
ActivityDiary	Off-Line	200ms	199	0	0	0	-	NullPointerException
			101	0	2	0	-	IndexOutOfBoundsException
			85	1	23	2	-	IndexOutOfBoundsException
			59	7	9	11	-	SQLException
		500ms	131	6	17	1	-	SQLException
			121	4	10	0	-	SQLException
			14	12	173	1	-	SQLException
			198	0	2	0	-	IndexOutOfBoundsException
			6	0	13	1	-	IndexOutOfBoundsException
AntennaPod	On-Line	500ms	0	0	0	0	0	VerifyError
			7	75	0	94	5	VerifyError
			61	0	91	0	2	VerifyError
			8	0	55	0	1	VerifyError
			27	12	145	8	8	libcoreError
			49	132	12	5	2	Ioexception
			48	87	33	9	5	InterruptedIOException
		1000ms	0	0	0	0	0	VerifyError
			37	93	59	8	2	IllegalArgumentException
			64	37	73	6	2	VerifyError
			0	8	0	2	1	VerifyError
NewPipe	On-Line	500ms	13	5	7	0	1	NullPointerException
		1000ms	187	4	6	0	0	RemoteServiceException
			167	15	8	0	0	NullPointerException
#Total			3864	1483	1071	653	29	
#Proportion			54.4%	20.9%	15.1%	9.2%	0.4%	

Fig. 5: Illustrative examples of root causes. The red boxes indicate the click area.

Dynamic Content Dynamic content refers to the situation where some specific app dynamic contents may change (e.g., recommended items, pop-up advertisements), leading to the different execution traces in replays from those in the original execution traces, thus resulting in the reproducibility issues of Monkey. In our experiments, 29 out of 7100 replays with reproducibility issues (accounting for 0.4%) are related to dynamic content.

In certain specific on-line apps, the presence of dynamic content introduces significant challenges to reproducing crashes. For example, in the case of *AntennaPod* illustrated in Fig.5(g) and (h), the continuous changes in the recommendation list primarily arise from the app's reliance on fetching and updating data from remote sources. User interactions and time-dependent factors trigger these data updates, resulting in constant changes in the recommendation list. Consequently, even though we use the read-only mode to ensure that the app starts from the same state every time, for apps with recommendation lists, the content of the recommendation list may change when run at different times. The dynamic nature of the recommendation lists may lead to discrepancies between the events executed in tau_O and tau_R.

4 Discussions and Implications

4.1 How does Throttle Affect Monkey's Reproducibility Rate?

A recent study by Feng *et al.*[8] proposed a lightweight image-based approach AdaT to dynamically adjust the inter-event time based on GUI rendering state. AdaT can infer the rendering state and synchronize with the testing tool to schedule the next event when the GUI is fully rendered, which can improve the testing effectiveness. Another study by Behrang *et al.* [3] also indicated that for UI-based flaky tests, the typical root causes behind flaky UI tests include issues like async wait and resource rendering due to improper configuration of time intervals between two events, leading to the flakiness of UI tests. According to our experimental results, we are curious about the impact of throttle on the reproducibility rate of Monkey. To investigate the relationship between the throttle and the reproducibility rate, we randomly chose 19 τ_O, both increased and decreased the throttle of each τ_O, and replayed them 200 times with the new throttle to get their corresponding new τ_R, then computed the new reproducibility rate. Then we similarly conducted the ANOVA analysis, and the results revealed the P-value of 0.280. This indicates that altering the throttle, whether increased or decreased, did not significantly improve the reproducibility rate. When the throttle is increased, the app gets a longer GUI loading time, but the reproducibility rate of the crash bug has not been significantly improved. One potential explanation for this phenomenon is that triggering a crash bug usually requires dozens of Monkey events to reach a specific state that leads to a crash. Some of the pivot events need to be executed rapidly, while others need to be executed after the app is fully rendered. So a uniform adjustment of throttle (whether increase or decrease) may potentially miss out on some pivot

events, making the app cannot reach the specific state to crash. Therefore, during testing, if each event waits until the GUI is fully loaded before execution, there's a possibility of missing some bugs which are triggered only when user events are executed during the partial rendering state of the GUI. Our experimental results indicate that a larger throttle isn't necessarily better. The better selection of intervals between events in automated GUI testing remains a topic worthy of discussion.

4.2 Can R&R Tools Improve Monkey's Reproducibility Rate?

After discovering the reproducibility issues of MONKEY, we wondered if the R&R (Record and Replay) tools could improve MONKEY's reproducibility rate. To validate this assumption, we initially selected three recent and representative R&R tools - RERAN[10], RX[14] and SARA[12]. Then we use MONKEY's built-in functionality to replay existing crash bugs, and record them with the R&R tools at the same time. However, we found that RERAN was unable to capture the events executed by MONKEY. This is because RERAN records events from the system file `/dev/input/event`. Only events captured by *InputReader* are logged into `/dev/input/event`. Consequently, the events generated by MONKEY cannot be recorded by RERAN. After that, we replayed the sequences recorded by RX and SARA and assessed their reproducibility rate. We conducted our small-scale experiments on two off-line apps namely *AmazeFileManager* and *AnkiDroid*, and selected two crash bugs with short τ_O and reproduced them five times with the R&R tools. According to our experimental results, we found that employing R&R tools to reproduce crash bugs yields a lower reproducibility rate than that of MONKEY. This is because MONKEY generates events quickly, and most of the R&R tools record events in the form of scripts, which is time-consuming. Secondly, the R&R tools can only record certain event types, so they cannot record all the events executed by MONKEY, which leads to a failure to reproduce the crash bug. This also highlights that for R&R tools, recording speed and comprehensive recording of event types are crucial and important.

4.3 Threats to Validity

Our study may suffer from some threats to validity. First, our research focused exclusively on MONKEY without assessing the reproducibility capabilities of other AIG tools. This is because MONKEY is a widely used AIG tool in the industry and is representative of commonly applied testing tools. Additionally, MONKEY itself provides self-replay capabilities, which eliminates the impact of additional record and replay tools on the experimental results. Moreover, many AIG tools (e.g., APE, FASTBOT) are designed upon MONKEY. Therefore, studying the reproducibility issues of MONKEY is a meaningful work and can provide insights to other AIG tools. Specifically, *Injection Failure* may apply to AIG tools that inject events into *InputManager*. *Data Loading* and *Widget Loading* may apply to AIG tools that are coordinate-sensitive but not widget-sensitive.

Event Ambiguity may not apply to widget-sensitive AIG tools, because they directly perform corresponding actions on the widgets. In the future, we plan to expand our research to investigate the reproducibility capabilities of other AIG tools as well. Second, our study involves some manual analysis, which may bring some potential biases in the results. To mitigate this threat, two co-authors independently conducted the analysis and then reached a consensus to derive the final results. When they could not reach a consensus, the other three co-authors participated and helped make the final decision together. This approach helps ensure a more objective and reliable assessment of the findings and minimizes the influence of individual biases.

Additionally, we have introduced InstrumentDroid, which may cause some potential problems. First, we detect bugs based on the instrumented app, which makes τ_O and τ_R unified. Second, InstrumentDroid only inserts a snippet of log code to the event handlers, which is a lightweight implant and will not have a big impact on the performance of the program. Moreover, we've verified InstrumentDroid's accuracy in event recognition, ensuring that the same UI controls don't produce duplicate content in *Event Handler LogFile*. Nevertheless, our tool does have its limitations. While it can cover most widgets, widgets without corresponding event handlers require special actions. Yet, this limitation minimally affects the reproducibility issues, as the behavior between the original and replay traces remains consistent.

5 Related Work

Flakiness in Android GUI Testing. Flaky tests refer to software tests that produce inconsistent or unreliable results. Different from the reproducibility issues, in the literature, flakiness usually refers to the uncertainty of test results. A flaky test does not necessarily trigger a crash bug. However, the reproducibility issues focus on a known bug and study whether the bug can be reliably reproduced. There are many works about the flakiness in UI tests and unit tests. Romano *et al.*[25] investigated flaky UI tests, identifying common causes such as Async Wait, Environment, Test Runner API, and Test Script Logic issues. SHAKER[27] offers a technique to improve test rerun effectiveness for spotting flaky tests. Both our study and previous ones found that UI rendering and asynchronous data loading contribute to flakiness. Our work uniquely introduces Injection Failure and Event Ambiguity as causes. Conversely, other studies highlight concurrency and thread competition as sources of flakiness.

Some works also researched the topic of reproducibility. Su *et al.* conducted a study about the reproducibility of exception bugs [28]. They chose two Android GUI Testing tools, i.e., Stoat [29] and Sapienz [19], and utilized Monkey and UIAutomator scripts for test recording and replay. If an app crashed, they recorded the exception trace and the crash-triggering test, rerunning each test five times to determine reproducibility. They identified three challenges for testing tools in reliably reproducing exceptions: test dependency, timing of events, and specific running environment. Our work differs in several respects. First, our

tool choice was Monkey due to its widespread industry use and built-in replay functionality, negating the need for extra scripts. Notably, Su *et al.* mentioned the flakiness of Monkey tests so they didn't choose it. Second, we replayed crash bugs 200 times for reproducibility, as opposed to their five times. Third, the 56 crash bugs in our work were discovered through random GUI testing using Monkey in a unified environment. These bugs are all independent of each other, so there is no correlation between them, and they are not affected by the testing environment. We also addressed event timing via *Data Loading* and *Widget Loading*. Compared to their work, our work is more systematic and comprehensive.

Deterministic Replay in Other Systems. Deterministic replay, often referred to as reproducibility, is less studied in the Android field than in non-smartphone platforms where it has been widely explored and implemented. In hardware, FDR [36] offers a low-overhead solution for reproducible execution in cache-coherent multiprocessors. Conversely, BugNet [21] is designed to record information continuously for deterministic bug replay and resolution. In virtual machines, ReVirt [7] enhances intrusion analysis by using logging and replay techniques, minimizing interference from the target OS. LoRe [15] serves a similar purpose, but is tailored for the popular full virtualization solution, KVM.

6 Conclusion and Future Work

In this paper, we conducted an in-depth empirical study on the reproducibility issues of Monkey about how effectively can it reproduce the crash bugs it detected and the root causes of its reproducibility issues. Specifically, we studied 56 unique crash bugs detected by Monkey from six popular open-source Android apps to understand the reproducibility issues. Our results show that only 36.6% of the crashes could be reproduced on average. Through the manual analysis, we categorized five types of root causes of the reproducibility issues of Monkey: *Injection Failure*, *Event Ambiguity*, *Data Loading*, *Widget Loading* and *Dynamic Content*. The corresponding proportions of them are 54.4%, 20.9%, 15.1%, 9.2%, and 0.4% on average. In the future, we plan to come up with some solutions to improve the reproducibility issues of Monkey and research the reproducibility issues of other AIG tools.

Acknowledgements. We thank the SETTA reviewers for their valuable feedback, Yiheng Xiong and Shan Huang from East China Normal University for their insightful comments, and Cong Li from Nanjing University for the mechanism of Rx. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002), NSFC Grant 62072178, "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, and the Shanghai Collaborative Innovation Center of Trusted Industry Internet Software.

References

1. Arnatovich, Y., Wang, L., Ngo, N., Soh, C.: Mobolic: An automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation: Mobolic: an automated approach to exercising mobile applications. Software: Practice and Experience **48** (02 2018). https://doi.org/10.1002/spe.2564
2. Ash Turner: The Rise of Android: Why is Android Successful? (2023), https://www.bankmycell.com/blog/how-many-android-users-are-there
3. Behrang, F., Orso, A.: Seven reasons why: An in-depth study of the limitations of random test input generation for android. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 1066–1077. ASE '20, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3324884.3416567
4. Bläsing, T., Batyuk, L., Schmidt, A.D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In: 2010 5th International Conference on Malicious and Unwanted Software. pp. 55–62 (2010). https://doi.org/10.1109/MALWARE.2010.5665792
5. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems **30**(19) (2002)
6. Chen, S., Fan, L., Su, T., Ma, L., Liu, Y., Xu, L.: Automated cross-platform gui code generation for mobile apps. In: 2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile). pp. 13–16 (2019). https://doi.org/10.1109/AI4Mobile.2019.8672718
7. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: Enabling intrusion analysis through virtual-machine logging and replay **36**(SI), 211–224 (dec 2003). https://doi.org/10.1145/844128.844148
8. Feng, S., Xie, M., Chen, C.: Efficiency matters: Speeding up automated testing with gui rendering inference. In: Proceedings of the 45th International Conference on Software Engineering. pp. 906–918. ICSE '23 (2023). https://doi.org/10.1109/ICSE48619.2023.00084
9. Girden, E.R.: ANOVA: Repeated measures. No. 84, Sage (1992)
10. Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: Timing- and touch-sensitive record and replay for android. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 72–81. IEEE Computer Society, Los Alamitos, CA, USA (may 2013). https://doi.org/10.1109/ICSE.2013.6606553
11. Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J., Su, Z.: Practical gui testing of android applications via model abstraction and refinement. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 269–280 (2019). https://doi.org/10.1109/ICSE.2019.00042
12. Guo, J., Li, S., Lou, J.G., Yang, Z., Liu, T.: Sara: Self-replay augmented record and replay for android in industrial cases. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330557
13. IBM Corp.: Ibm spss statistics for windows, https://hadoop.apache.org
14. Li, C., Jiang, Y., Xu, C.: Cross-device record and replay for android apps. pp. 395–407. ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3540250.3549083
15. Li, J., Si, S., Li, B., Cui, L., Zheng, J.: Lore: Supporting non-deterministic events logging and replay for kvm virtual machines. In: 2013 IEEE 10th International

Conference on High Performance Computing and Communications. vol. 1, pp. 442–449 (2013). https://doi.org/10.1109/HPCC.and.EUC.2013.70
16. Li, Y., Yang, Z., Guo, Y., Chen, X.: Droidbot: a lightweight ui-guided test input generator for android. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 23–26 (2017). https://doi.org/10.1109/ICSE-C.2017.8
17. Li, Y., Yang, Z., Guo, Y., Chen, X.: Humanoid: A deep learning-based approach to automated black-box android app testing. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1070–1073 (2019). https://doi.org/10.1109/ASE.2019.00104
18. Lv, Z., Peng, C., Zhang, Z., Su, T., Liu, K., Yang, P.: Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22 (2023), https://doi.org/10.1145/3551349.3559505
19. Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ISSTA 2016 (2016). https://doi.org/10.1145/2931037.2931054
20. Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., Poshyvanyk, D.: Automatically discovering, reporting and reproducing android application crashes. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 33–44 (2016). https://doi.org/10.1109/ICST.2016.34
21. Narayanasamy, S., Pokam, G., Calder, B.: Bugnet: Continuously recording program execution for deterministic replay debugging. pp. 284–295. ISCA '05, IEEE Computer Society, USA (2005). https://doi.org/10.1109/ISCA.2005.16
22. Patel, P., Srinivasan, G., Rahaman, S., Neamtiu, I.: On the effectiveness of random testing for android: Or how i learned to stop worrying and love the monkey. In: Proceedings of the 13th International Workshop on Automation of Software Test. pp. 34–37 (2018). https://doi.org/10.1145/3194733.3194742
23. Project, A.O.S.: Monkey - android developers (2023), https://developer.android.com/studio/test/other-testing-tools/monkey
24. Project, A.O.S.: Sdk platform tools release notes (2023), https://developer.android.com/tools/releases/platform-tools
25. Romano, A., Song, Z., Grandhi, S., Yang, W., Wang, W.: An empirical analysis of ui-based flaky tests. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1585–1597 (2021). https://doi.org/10.1109/ICSE43902.2021.00141
26. Roy Choudhary, S., Gorla, A., Orso, A.: Automated test input generation for android: Are we there yet? (e). pp. 429–440 (11 2015). https://doi.org/10.1109/ASE.2015.89
27. Silva, D., Teixeira, L., d'Amorim, M.: Shake it! detecting flaky tests caused by concurrency with shaker. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 301–311 (2020). https://doi.org/10.1109/ICSME46990.2020.00037
28. Su, T., Fan, L., Chen, S., Liu, Y., Xu, L., Pu, G., Su, Z.: Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps. IEEE Transactions on Software Engineering **48**(4), 1115–1137 (2022). https://doi.org/10.1109/TSE.2020.3013438
29. Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z.: Guided, stochastic model-based gui testing of android apps. In: Proceedings of the

2017 11th Joint Meeting on Foundations of Software Engineering. pp. 245–256. ESEC/FSE 2017 (2017). https://doi.org/10.1145/3106237.3106298
30. Su, T., Wang, J., Su, Z.: Benchmarking automated gui testing for android against real-world bugs. In: Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 119–130 (2021). https://doi.org/10.1145/3468264.3468620
31. Su, T., Yan, Y., Wang, J., Sun, J., Xiong, Y., Pu, G., Wang, K., Su, Z.: Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs **5**(OOPSLA) (2021), https://doi.org/10.1145/3485533
32. Sun, J., Su, T., Li, J., Dong, Z., Pu, G., Xie, T., Su, Z.: Understanding and finding system setting-related defects in android apps. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 204–215 (2021), https://doi.org/10.1145/3460319.3464806
33. Wang, J., Jiang, Y., Xu, C., Cao, C., Ma, X., Lu, J.: Combodroid: Generating high-quality test inputs for android apps via use case combinations. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 469–480. ICSE '20 (2020), https://doi.org/10.1145/3377811.3380382
34. Wang, W., Li, D., Yang, W., Cao, Y., Zhang, Z., Deng, Y., Xie, T.: An empirical study of android test generation tools in industrial cases. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 738–748. ASE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3240465
35. Xiong, Y., Xu, M., Su, T., Sun, J., Wang, J., Wen, H., Pu, G., He, J., Su, Z.: An empirical study of functional bugs in android apps. pp. 1319–1331 (2023). https://doi.org/10.1145/3597926.3598138
36. Xu, M., Bodik, R., Hill, M.D.: A "flight data recorder" for enabling full-system multiprocessor deterministic replay. pp. 122–135. ISCA '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/859618.859633