# A Fast Crash Reproduction Method for Android Applications Based on Widget Hierarchy Graphs

Zhanqi Cui, *Member, IEEE*, Gaoyi Lin, Liwei Zheng, and Zhihua Zhang

***Abstract*—To improve the efficiency of fixing bugs, mobile application developers must reproduce bugs reported by testers or users as quickly as possible. In some cases, automated testing tools can help developers reproduce crashes. However, these tools were not designed for reproducing bug reports. They are not efficient at reproducing crashes. To focus testing resources on suspicious widgets, we propose CrPDroid, a fast crash reproduction method for Android applications based on widget hierarchy graphs. First, it builds a widget hierarchy graph by using the project file of the application under test; then, it locates suspicious widgets by analyzing the bug report and the project file of the application under test and calculates the fitness of each widget according to the widget hierarchy graph; finally, it uses the fitness of widgets to guide automated testing to reproduce crashes quickly. To evaluate the effectiveness of CrPDroid, experiments are conducted on real Android application bug reports, and the crash reproduction tool ReCDroid, ReproBot and automated testing tools APE and PUMA are compared with CrPDroid. The experimental results show that CrPDroid successfully reproduces 15 bug reports that cause Android app crashes. In addition, compared with APE, PUMA, ReCDroid, and ReproBot, the average time for CrPDroid to reproduce crashes decreased by 76.87%, 81.94%, 95.58%, and 76.55%, and the total number of operations on suspicious widgets in the same period of testing time increased by 44.07%, 87.57%, 88.70%, and 68.93% on average, respectively.**

***Index Terms*—Android applications, bug reports, reproduce crashes, widget hierarchy graphs.**

## I. Introduction

With the rapid development of 5G [1], Internet of Things (IoT) devices are penetrating various areas of our lives [2]. The number of IoT devices based on the Android platform, especially Android smartphones, is experiencing explosive growth [3]. As the Android system continues to grow in popularity, Android mobile applications have also proliferated. In 2022, Google Play released more than 2.6 million Android applications [4]. Facing fierce competition in the mobile application market, developers must release new versions of applications as quickly as possible to remain competitive. As a result, it is difficult to test mobile applications thoroughly, and many bugs are still contained in the release versions. Furthermore, this situation leads to increased testing and maintenance costs and challenges the robustness and reliability of mobile applications.

It has been found that 88% of users abandon mobile applications when encountering bugs repeatedly [5]. Therefore, bugs in mobile applications result in user loss, and developers must respond quickly to the bugs found and fix them, especially application crashes, which directly affect the usability of mobile applications [6]. To debug and fix crashes, it is first necessary to reproduce crashes. Reproducing crashes may require complex operations since mobile applications typically involve many interfaces and executable events. Therefore, manually reproducing crashes is inefficient. Automated testing tools can assist developers in reproducing crashes, but they are less efficient. For example, Google's automated testing tool Monkey [7] can generate random UI events and inject them into Android applications, regardless of their design details. Even though such automated testing tools can trigger some crashes in applications, their search scope contains all executable paths of the application, including irrelevant ones, so they are less efficient in reproducing crashes.

Software projects use bug-tracking systems (such as Bugzilla,[1] Google Code Issue Tracker,[2] GitHub Issue Tracker,[3] etc.) to manage the testing process and accelerate bug fixing. Such bug-tracking systems allow testers to report bugs during testing. Additionally, users can report bugs on the mobile app market (e.g., Google Play)[4] by posting comments. Reviewing bug reports in the bug-tracking system and user comments is an important way to find and reproduce crashes. When reporting bugs, users and testers usually provide application versions, system versions, bug screenshots, stack traces, widget information, etc. By using the stack trace and widget information in the bug report to reproduce a crash, a developer can refine the search scope when manually locating the crash, focus on the widget or function that caused the crash,

Manuscript received 25 September 2023; revised 11 December 2023; accepted 16 January 2024. Date of publication 22 January 2024; date of current version 9 April 2024. This work was supported in part by the Jiangsu Provincial Frontier Leading Technology Fundamental Research Project under Grant BK20202001; in part by the Open Research Fund of Key Laboratory of Safety-Critical Software Fund (Nanjing University of Aeronautics and Astronautics) under Grant NJ2023031; in part by the National Natural Science Foundation of China under Grant 61702041; and in part by the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project under Grant QXTCP C201906. *(Corresponding author: Zhanqi Cui.)*

Zhanqi Cui and Gaoyi Lin are with the Computer School, Beijing Information Science and Technology University, Beijing 100101, China (e-mail: czq@bistu.edu.cn; lingaoyi1998@bistu.edu.cn).

Liwei Zheng and Zhihua Zhang are with the Computer School, Beijing Information Science and Technology University, Beijing 100101, China (e-mail: zlw@bistu.edu.cn; zhangzh@bistu.edu.cn).

Digital Object Identifier 10.1109/JIOT.2024.3357209

[1]Bugzilla. https://bugzilla.mozilla.org/describekeywords.cgi.
[2]Google Code Issue Tracker. https://code.google.com.
[3]GitHub Issue Tracker. https://github.com.
[4]Google Play. https://play.google.com/store/apps.


Fig. 1. Example bug report for BetterBatteryStats.

and allocate more testing resources to the locations related to the bug report to improve the efficiency of reproducing the crash. For example, when a developer receives the bug report shown in Fig. 1, they can focus on the "Show only %" widget. However, existing automated testing tools rarely consider bug reports. Furthermore, even if the widget that caused the application crash is found, the application, which consists of many interfaces and executable events, needs to be executed following specific operation sequences to reproduce the crash. As shown in Fig. 2, to reproduce the crash described in the bug report in Fig. 1, four steps need to be executed in sequence: 1) click "more options" as shown in Fig. 2(a); 2) click "Settings" as shown in Fig. 2(b); 3) scroll down as shown in Fig. 2(c); and 4) click "Show only %" as shown in Fig. 2(d).

As the example shows, it is possible to increase the efficiency of reproducing a crash by using bug report information effectively and concentrating more testing resources on suspicious widgets. Some works [8], [9], [10], [11] focused on reproducing crashes with bug reports. Taking two state-of-the-art works, ReproBot [10] and ScopeDroid [11], as examples, their primary insight is extracting crash reproduction steps from the natural language in bug reports and attempting to execute the applications under test according to the extracted steps to reproduce the crashes. However, a large proportion of bug reports provide incomplete crash reproduction steps, making it challenging to reproduce crashes efficiently. In extreme cases, these approaches are degraded to general automation testing tools. Moreover, they do not consider other useful information provided in the bug reports, such as stack traces. To mitigate this issue, we propose crash reproduction method for Android applications (CrPDroid), a fast crash reproduction method for Android applications based on widget hierarchy graph. CrPDroid analyzes the project files of the application under test to construct a widget hierarchy graph that records the relationships between widgets as well as the relationships between widgets and functions. In combination with the crash reproduction steps and stack traces in the bug report, CrPDroid can find the path for crash reproduction based on the graph more efficiently. First, CrPDroid analyzes the project file of the application under test and builds a widget hierarchy graph based on function calls, interface jumps, and relationships between widgets and functions. Second, it locates suspicious widgets by analyzing bug reports and project files and calculates the fitness of each widget by using the widget hierarchy graph. Third, the fitness of the widget is used to guide automated testing to reproduce the crash quickly. Previously, we proposed the technical framework of this method and conducted preliminary experiments [12]. In this article, we expand more technical details of the crash reproduction method, including the construction of the widget hierarchy graph, the fitness calculation of widgets, and crash reproduction with the fitness of widgets. In addition, we expand the experiments to further evaluate the effectiveness of CrPDroid. Specifically, the scale of the representative experimental subjects is expanded from 6 to 16. Moreover, two more state-of-the-art tools are introduced for comparison. We also conduct experiments to discuss the impact of the fitness calculation parameters and the widget hierarchy graph on the performance of CrPDroid in terms of testing resource allocation. The experimental results show that CrPDroid, APE [13], PUMA [14], ReCDroid [8], and ReproBot [10] successfully reproduce 15, 14, 9, 7, and 11 crashes, respectively. Compared with PUMA, CrPDroid saves 81.94% of the time of reproducing crashes on average and performs 87.57% more operations on suspicious widgets in the same period of testing time. Compared with APE, CrPDroid saves 76.87% of the time needed to reproduce crashes on average and performs 44.07% more operations on suspicious widgets. Compared with ReCDroid, CrPDroid saves 95.58% of the time needed to reproduce crashes on average and performs 88.70% more operations on suspicious widgets. Compared with ReproBot, CrPDroid saves 76.55% of the time needed to reproduce crashes on average and performs 68.93% more operations on suspicious widgets.

This article makes the following contributions.

1) We propose CrPDroid, which is a fast crash reproduction method for Android applications based on widget hierarchy graphs, to improve the efficiency of reproducing crashes. In contrast to works focusing on the quality of bug reports, CrPDroid aims to directly reproduce the crashes described in the bug reports.
2) We propose widget hierarchy graphs, which describe the relationships between widgets. The experimental results show that the graphs can be used to guide dynamic exploration for rapid reproduction of crashes.
3) Experiments are carried out on a set of real Android applications, and the experimental results show that CrPDroid outperforms the automated testing tools APE and PUMA, as well as the crash reproduction tool ReCDroid, in terms of crash reproduction quantity, crash reproduction efficiency, and testing resource allocation.

The remainder of this article is organized as follows. Section II presents the details of the fast crash reproduction method for Android applications based on widget hierarchy graphs. Section III presents the experimental design and results. Section IV discusses the limitations and threats to the validity of the method. Section V introduces related work. Section VI presents the conclusion.

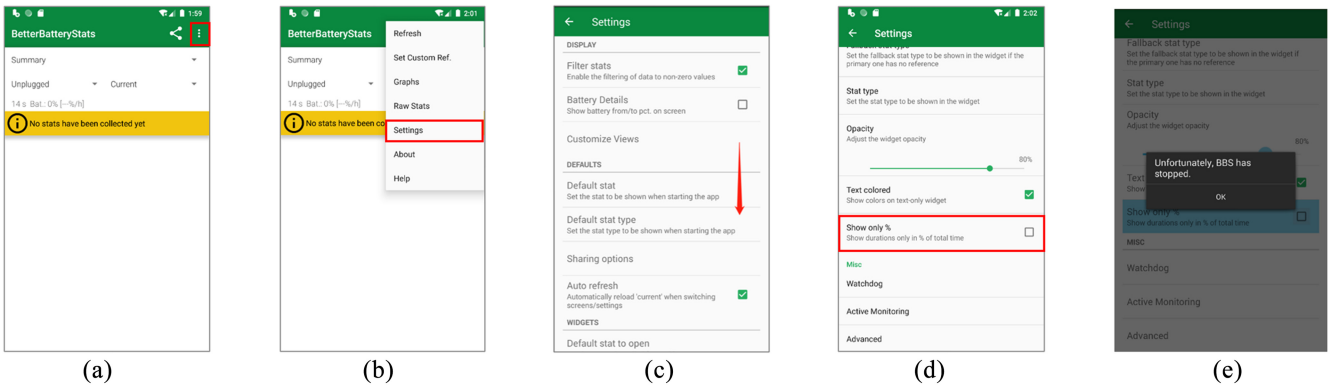


Fig. 2. Steps for reproducing the crash described in the bug report shown in Fig. 1. (a) Click "more options". (b) Click "Settings". (c) Scroll down. (d) Click "Show only %". (e) Crash.

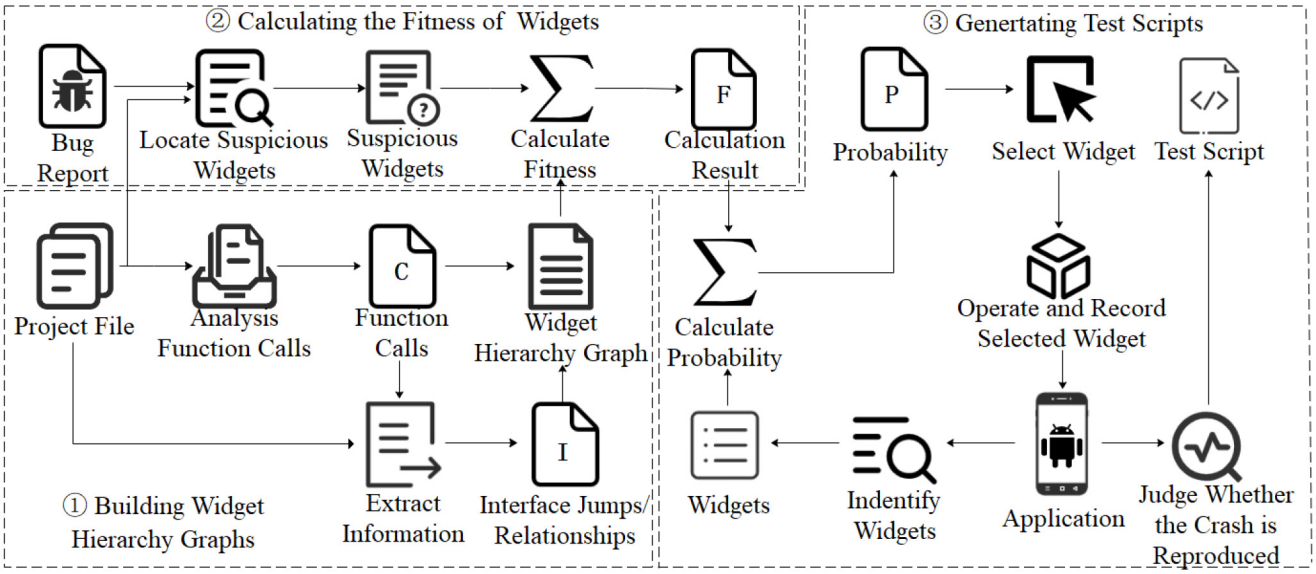


Fig. 3. Framework of the fast crash reproduction method for Android applications based on widget hierarchy graphs.

## II. Fast Crash Reproduction Method for Android Applications Based on Widget Hierarchy Graphs

Fig. 3 shows the framework of the fast crash reproduction method for Android applications based on widget hierarchy graphs. First, it analyzes and obtains the function calls from the project file of the application under test and extracts the interface jumps and the relationships between widgets and functions from the project file in combination with the function calls to complete the creation of the widget hierarchy graph. Then, it locates suspicious widgets by using the bug report and the project file and calculates the fitness of widgets by using the suspicious widgets and the widget hierarchy graph. Third, it uses the fitness of widgets to guide automated testing and outputs test scripts for reproducing the crash.

### A. Building Widget Hierarchy Graphs

We use the widget hierarchy graph to describe the relationships between widgets, as well as the relationships between widgets and functions of Android applications. A widget hierarchy graph consists of nodes corresponding to functions in the Android application and edges corresponding to function calls or interface jumps in the Android application. A widget hierarchy graph can be defined as follows.

*Definition 1 (Widget Hierarchy Graph):* A widget hierarchy graph of Android application $A$ is a 5-tuple $G = (V, C, J, W, P)$, where:

1) $V = \{v_1, v_2, \ldots, v_i, \ldots\}$ is the finite set of nodes in the graph, and node $v_i \in V$ is a function in $A$;
2) $C = \{c_{1,2}, c_{3,4}, \ldots, c_{k,l}, \ldots\}$ and $J = \{j_{1,2}, j_{3,4}, \ldots, j_{m,n}, \ldots\}$ are finite sets of two kinds of edges in the graph; the edge $c_{k,l} \in C$ is a function call in $A$, and the edge $j_{m,n} \in J$ is an interface jump in $A$;
3) $W$ is the finite set of widgets in $A$;
4) $P$ is the finite set of interfaces in $A$, and each interface consists of several widgets.

For the function $v_i \in V$, $W_i = Annotation(v_i) \subseteq W$ is the set of widgets related to the function $v_i$, indicating that the function $v_i$ will be called when operating any widget in $W_i$; if $Annotation(v_i) = \emptyset$, then the function $v_i$ has no related widgets. For the function call $c_{k,l} \in C$, $Caller(c_{k,l}) = v_k \in V$, $Callee(c_{k,l}) = v_l \in V$, indicating that the function $v_k$ will call the function $v_l$. For the interface jump $j_{m,n} \in J$, $From(j_{m,n}) =$

$p_m \in P$ and $To(j_{m,n}) = p_n \in P$, indicating that interface $p_m$ can jump to interface $p_n$, where the starting point of the jump is $p_m$ and the ending point of the jump is $p_n$. Trigger$(j_{m,n}) = w_v^m \in p_m$, indicating that the widget $w_v^m$ in the interface $p_m$ is the widget that triggers the interface jump to $p_n$.

To build the widget hierarchy graph, CrPDroid first needs to get the set of functions $V$ and the set of function calls $C$ to construct the graph. Specifically, CrPDroid compiles the project file of the application under test to an APK file and analyzes the APK file to get function calls. To obtain function calls, CrPDroid uses FlowDroid[5] to perform static taint tracking analysis [15]. The calling functions and the called functions in function calls are both added to the set of functions $V$. The relationships between functions in terms of calls are recorded in the set of function calls $C$. In Android applications, function calls differ from those in ordinary programs. An Android application is composed of components with complex lifecycles (Activity, Service, BroadcastReceiver, and ContentProvider) and developer-defined GUI callback methods. Among them, components with complex lifecycles provide multiple callback methods (i.e., lifecycle callback methods [16]). By calling these lifecycle callback methods, components can transit between different states of lifecycles. The Android framework calls these lifecycle callback methods and GUI callback methods implicitly. FlowDroid automatically generates dummy methods to represent implicit invocations of lifecycle and GUI callback methods in a specific order to reduce the complexity of analyzing Android applications.

After obtaining the set of functions $V$ and the set of function calls $C$, CrPDroid analyzes the project file of the application under test in combination with function calls to obtain interface jumps and the relationships between widgets and functions. First, get the set of widgets $W$ from the layout files of the application under test. The layout files record widgets and their attributes (including android:id, android:text, android:contentDescription, etc.). CrPDroid identifies widgets in the layout files and adds them to the set of widgets $W$. Then, obtain the functions related to the widgets in $W$, i.e., event handling functions, when operating one widget, the widget will automatically call the event handling function associated with it. The relationships between functions and widgets can be obtained by static analysis tool Gator,[6] which is a program analysis toolkit for Android. For the function $v_i \in V$, $W_i = Annotation(v_i) \in W$, that is, the function $v_i$ and the widgets in $W_i$ are related to each other. Next, we traverse the set of functions $V$. If $v_u$ is a dummy function, we obtain the set of widgets $W_u = Annotaion(v_u) \subseteq W$ related to the functions called by $v_u$ and add the interface $p_u$ in which $W_u$ is located to the interface set $P$. Finally, identify the interface jumps of the application by using Gator. If interface $p_m$ can jump to interface $p_n$, we add interface jump $j_{m,n}$ to $J$, where From$(j_{m,n}) = p_m$, $To(j_{m,n}) = p_n$.

Fig. 4 shows a fragment of the widget hierarchy graph of the BetterBatteryStats application. In Fig. 4, nodes represent functions, and solid arrows represent function calls. For example, function 5 is called by function 2. Some functions have widgets related to them. For example, "Text colored," "Show only %," and other widgets are related to function 7. Nodes 2 and 3 represent dummy functions, and all widgets related to functions called by dummy functions are within the same interface; for example, widgets "Refresh" and "Settings" are all within the same interface. The dotted arrows between dummy functions 2 and 3 indicate an interface jump, which is triggered by the widget "Settings."

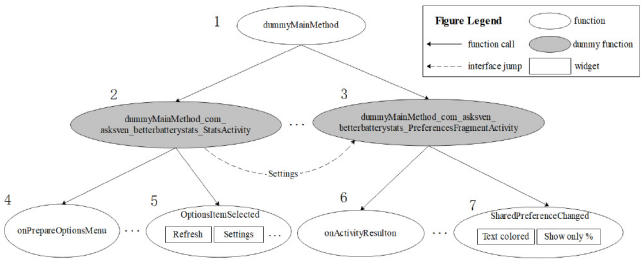


Fig. 4. Widget hierarchy graph fragment for the BetterBatteryStats application.

## B. Calculating the Fitness of Widgets

The fast crash reproduction method for Android applications based on widget hierarchy graphs guides automated testing by the fitness of each widget. The fitness of widgets measures whether a widget is likely to trigger a crash. The greater the fitness value of a widget, the more likely it is to trigger a crash. To determine the fitness of widgets, CrPDroid also needs to obtain the set of suspicious widgets $W_{\text{susp}}$.

The set of suspicious widgets $W_{\text{susp}}$ is obtained from the project file of the application under test, where CrPDroid uses the crash reproduction steps described in natural language and the stack trace contained in the bug report to find the widgets related to the bug report. Specifically, for the part of the bug report that is written in natural language describing the crash reproduction steps, CrPDroid first use the dependency analysis tool spaCy [17] to divide the crash reproduction steps into sentences, and apply stemming [18] to get the root forms of the words in each sentence. Then, words describing widgets are extracted from the sentences (the stemmed words). In the work of Zhao et al. [8], they manually analyzed 813 bug reports to construct grammar patterns. CrPDroid adapts the grammar patterns[7] to extract words describing widgets from sentences. Finally, CrPDroid iterately checks all widgets in the set $W$, checking whether they match the extracted words describing the widgets. If a match is found, the widget is added to the suspicious widget set $W_{\text{susp}}$. The Word2Vec [19] model is used for the matching by calculating the semantic similarity (a score within the range of [0, 1]) between the words describing the widgets and the widgets (using the android:text and android:contentDescription attributes).

For the stack trace in the bug report, CrPDroid extracts the first frame [20] in the stack (i.e., the function that causes the

[5]FlowDroid. https://blogs.uni-paderborn.de/sse/tools/flowdroid/.
[6]Gator. http://web.cse.ohio-state.edu/presto/software/gator/.
[7]Grammar Patterns. https://github.com/AndroidTestBugReport/ReCDroid/tree/master/nlp%20pattern.

**Algorithm 1** Calculate the Fitness of Widgets

**Input** Set of suspicious widgets $W_{\text{susp}}$, widget hierarchy graph $G = (V, C, J, W, P)$
**Output** Fitness $F$

1: $F \leftarrow \emptyset$
2: $P_{locate} \leftarrow$ FindPageOfSuspWidget($W_{\text{susp}}$, $G$)
3: **for** each page $p_n \in P_{locate}$ **do**
4: **for** each widget $w_u^n \in p_n$ **do**
5: $F$.add($w_u^n$, CalFitWithFormula1($w_u^n$, $W_{\text{susp}}$))
6: **while** first cycle **or** $p_{start} \neq \emptyset$ **do**
7: $p_{start} \leftarrow$ FindStartingPageOfJump($P_{locate}$, $G$)
8: **if** $P_{start} \neq \emptyset$ **then**
9: **for** each page $p_m \in P_{start}$ **do**
10: **for** each widget $w_v^m \in p_m$ **do**
11: $F$.add($w_v^m$, CalFitnWithFormula2($w_v^m$, $G$))
12: $P_{locate} = P_{start}$
13: **return** $F$

crash), and checks whether this function exists in the function set $V$. If so, it then checks whether the function can be invoked by operating widgets in the widget set $W$. If the function can be invoked by operating a widget, the widget is added to the suspicious widget set $W_{\text{susp}}$. Specifically, CrPDroid analyzes the function call chains within the set of function calls $C$. If one function call chain contains the first frame in the stack, and there is a function in the chain is related to a widget, the widget is considered a suspicious widget.

After getting the set of suspicious widgets $W_{\text{susp}}$, CrPDroid first finds the interfaces to which each suspicious widget belongs to and calculates the fitness of each widget in these interfaces; then, it finds the starting interfaces of jumps that take the above interfaces as the ending points of the jumps and calculates the fitness of each widget in the starting interfaces of the jumps.

Algorithm 1 is the algorithm for calculating the fitness of widgets. The input is the set of suspicious widgets $W_{\text{susp}}$ and the widget hierarchy graph $G = (V, C, J, W, P)$, and the output $F$ is the fitness values of the widgets in $W_{\text{susp}}$. After initializing the set $F$ (line 1) used to save the fitness values, CrPDroid traverses all the interfaces in the widget hierarchy graph, finds the interfaces to which all suspicious widgets belong, and records the set of interfaces $P_{\text{locate}}$ that contains suspicious widgets (line 2). Lines 3–5 traverse the widgets in each interface in $P_{\text{locate}}$, calculate the fitness of these widgets, and save the results in $F$. Next, CrPDroid continuously searches the set of starting points of interface jumps and calculates the fitness of widgets at the starting points (lines 6–12) until the set is empty. In each cycle, CrPDroid first traverses all interface jumps in the widget hierarchy graph, finds the interfaces that take any interface in the set of interfaces built in the previous iteration as the ending points of interface jumps, and saves the found interfaces as the set $p_{\text{start}}$ (line 7). If $p_{\text{start}}$ is not empty, CrPDroid traverses the widgets in each interface of $p_{\text{start}}$ and then takes the interfaces in $p_{\text{start}}$ as the ending points of the jumps to proceed with the next iteration after calculating the fitness of the widgets (lines 9–11); if $p_{\text{start}}$ is empty, indicating that there is no interface that is the starting point of an interface belonging to the $p_{\text{start}}$ set created by the previous iteration, the algorithm ends the iteration and exports $F$, which saves the fitness values of the widgets (line 13).

For the interface $p_n \in P_{\text{locate}}$ to which the suspicious widget belongs (line 2), CrPDroid uses (1) to calculate the fitness of each widget in $p_n$ (line 5). To make a suspicious widget in $p_n$ more likely to be covered during testing, for the widget $w_u^n \in p_n$, if $w_u^n \in W_{\text{susp}}$, then its fitness is $K \times N$; if $w_u^n \notin W_{\text{susp}}$, then its fitness is $N$. Here, $N$ is a nonzero constant, and $K$ is a constant greater than 1

$$\text{Fit}(w_u^n) = \begin{cases} K \times N, & \text{if } w_u^n \in W_{\text{susp}} \text{ and } w_u^n \in p_n \\ N, & \text{if } w_u^n \notin W_{\text{susp}} \text{ and } w_u^n \in p_n. \end{cases} \quad (1)$$

For interface $p_m \in P_{\text{start}}$ (line 7), we have $p_n \in P_{\text{locate}}$ and $j_{m,n} \in J$ such that $\text{From}(j_{m,n}) = p_m$ and $To(j_{m,n}) = p_n$. CrPDroid uses (2) to calculate the fitness of widgets in $p_m$. Widgets that can trigger interface jumps may lead the testing to the interface that triggers the application crash. To make such widgets more likely to be covered during testing, for $w_v^m \in p_m$, if $w_v^m$ is a widget in $p_m$ that can trigger the interface jump to $p_n$, its fitness is the sum of the fitness values of all widgets in $p_n$, that is, $\sum_{w_u^n \in p_n} \text{Fit}(w_u^n)$; if $w_v^m$ in $p_m$ is not a widget that can trigger an interface jump, its fitness is $N$

$$\text{Fit}(w_v^m) = \begin{cases} \sum_{w_u^n \in p_n} \text{Fit}(w_u^n), & \text{if Trigger}(j_{m,n}) = w_v^m \text{ and } w_v^m \in p_m \\ N, & \text{if Trigger}(j_{m,n}) \neq w_v^m \text{ and } w_v^m \in p_m. \end{cases} \quad (2)$$

Still taking the BetterBatteryStats application as an example, use Algorithm 1 and the widget hierarchy graph to calculate the fitness of its widgets. According to the bug report, the suspicious widget is "Show only %." The fitness of the suspicious widget "Show only %" is $K \times N$, and the fitness of the widget "Text colored" in the same interface is $N$ by using (1). In addition, the widget "Settings" is the widget that can trigger the interface to jump to the interface where the suspicious widget is located. By using (2), the fitness of the widget "Settings" is the sum of the fitness values of all widgets in the interface where the suspicious widget "Show only %" is located, that is, $K \times N + (s-1) \times N$ ($s$ is the number of widgets in the interface where "Show only %" is located), while the fitness of the widget "Refresh" in the same interface is $N$.

### C. Generating Test Scripts

When the calculation of fitness for widgets is completed, automated testing can be guided by the fitness values to reproduce the crashes more quickly. During testing, CrPDroid continuously selects and operates widgets in the current interface of the application based on the fitness of widgets. In each cycle, CrPDroid first calculates the probability of each widget being selected based on its fitness in the current interface and then selects and operates the widget according to the probability. According to (1) and (2), all widgets in the same interface are given appropriate fitness, giving each widget a certain probability of being selected. This is because triggering an application crash may require operating

**Algorithm 2** Fitness-Guided Automated Testing

**Input** Application under test $A$, fitness $F$, maximum test time *LIMIT*

**Output** Test script $R$

```
1: Launch A, R ← ∅
2: while time < LIMIT do
3:     Page ← GetCurrentPage(A)
4:     if triggers BugReport crash then
5:         return R
6:     if isFullyExpl(Page) then
7:         Execute(BACK, A)
8:         R.add(BACK)
9:         Continue
10:    ListofWidget ← GetAllWidgets(Page)
11:    ProbOfWidgetSelected ← CalProbWithFormula3(Lis-
12: tofWidget, F)
13:    w_select ← SelectWidget(ListofWidget,
    ProbOfWidgetSelected)
14:    Execute(w_select, A)
15:    R.add(w_select)
16: return null
```

widgets other than the suspicious widgets mentioned in the bug report. CrPDroid ends the testing when the maximum test time is exceeded or the application crash is successfully reproduced and then outputs the test script. It should be noted that CrPDroid does not strive to reproduce crashes strictly according to the steps described in the bug report, as the crash reproduction steps provided by users are not always accurate. When a crash occurs during the exploration, CrPDroid compares the system logs with the stack trace in the bug report. If the bug information of the system log is consistent with the stack trace in the bug report, the crash is considered successfully reproduced, otherwise, the exploration process continues. As CrPDroid obtains the widget hierarchy graph by statically analyzing the project file of the application under test, it may miss some widgets. To avoid the situation in which some widgets are never selected for operation because they are not identified by the static analysis, if a widget not identified by static analysis is encountered during the testing, its fitness will be set to the nonzero constant $N$.

Algorithm 2 describes the algorithm of the fitness-guided automated testing. The input is the application under test $A$, the fitness of widgets $F$, and the maximum test time *LIMIT*. The output is a testing script $R$ that can reproduce the crash. In line 1, CrPDroid launches the application under test and initializes the test script $R$. Then, CrPDroid starts to continuously explore the application under test (lines 2–14). When the exploration time exceeds the maximum test time (line 2) or the crash has been successfully reproduced (lines 4 and 5), the exploration is terminated. During the exploration process, line 3 obtains the current interface information of $A$. After identifying all widgets from the current interface (line 10), CrPDroid calculates the probability of each widget being selected according to the fitness of the widgets (line 11). In lines 12 and 13, CrPDroid selects and operates a widget in the current interface according to the probability and records the operation in the test script $R$ (line 14). To avoid overexploration of a certain interface during the testing, CrPDroid will judge whether the current interface has been fully explored before further operating on the interface (line 6). If the current interface has been fully explored, CrPDroid will return to the previous interface to explore. Here, the condition for judging whether the interface is fully explored is that all widgets in the interface have been operated on and the total number of operations on the widgets in the current interface is greater than $L$.

The fitness of a widget directly affects the probability of the widget being selected. In line 11, CrPDroid uses (3) to calculate the probability that each widget in $p_t$ is selected to be operated on. Among them, $\mathrm{Fit}(w_y^t)$ is the fitness of widget $w_y^t$ in interface $p_t$, and $\sum_{w_z^t \in p_t} \mathrm{Fit}(w_z^t)$ is the sum of the fitness values of all widgets in $p_t$. For widget $w_y^t \in p_t$, the probability of being selected to be operated on is the ratio of its fitness to the sum of the fitness values of all widgets in $p_t$

$$Pb\left(w_y^t\right) = \frac{\mathrm{Fit}\left(w_y^t\right)}{\sum_{w_z^t \in p_t} \mathrm{Fit}\left(w_z^t\right)}. \tag{3}$$

## III. Experiments and Evaluations

### A. Experimental Design

*1) Research Questions:* Four RQs are raised to evaluate the effectiveness of CrPDroid.

*RQ1: What impact does fitness calculation parameter have on the ability of CrPDroid to reproduce crashes?*

The parameter $K$ affects the fitness of widgets calculated by CrPDroid. Therefore, how does the parameter $K$ affect the ability of CrPDroid to reproduce crashes?

*RQ2: How does widget hierarchy graph guidance affect the performance of CrPDroid?*

The widget hierarchy graph guides the automated testing process of CrPDroid, so how does this graph affect the performance of CrPDroid?

*RQ3: In comparison with other automated testing tools for Android applications, how effectively and efficiently does CrPDroid reproduce crashes?*

CrPDroid aims to use the information in the bug report to improve the efficiency of crash reproduction, so how effective and efficient is CrPDroid compared to other automated testing tools?

*RQ4: Can CrPDroid concentrate more testing resources on suspicious widgets than other automated testing tools for Android applications?*

To increase the efficiency of crash reproduction, CrPDroid uses information from the bug report to reduce the number of unnecessary paths to be explored. Compared to other automated testing tools, is CrPDroid able to concentrate more resources on testing suspicious widgets?

*2) Experimental Settings:* In the experiments for RQ1–RQ3, we follow the experimental settings of Zhao et al. [8] and limit the testing time of each experiment to 2 h. For RQ4, the execution time of each experiment is 30 min. To reduce the impact of randomness, each testing tool is set to run ten

times on each experimental subject. The average of the results is recorded as the final time. Furthermore, the parameter $L$, which is used to evaluate whether the interface has been fully explored, is set to 100 in the experiments. And adopting the settings of ReCDroid, the threshold value of semantically similar words is set to great than or equal to 0.8.

*3) Experimental Subjects:* The experimental subjects used in this article are taken from Q-testing [21] and ReCDroid [8]. In Q-testing, Pan et al. [21] analyzed the standard benchmark in evaluating automated testing tools for Android applications [22], [23], removed outdated applications, and obtained 34 applications that can be found on F-Droid or GitHub. Furthermore, to enrich the test objects, they randomly selected several different categories of applications from the open-source application list [24] and finally obtained a benchmark consisting of 50 Android applications from F-Droid and GitHub. We analyze the above 50 applications collected by Q-testing. First, we search for "Crash" on the bug-tracking systems of the 50 applications, and 27 bug reports are collected. Then, the crashes described in the bug reports are manually verified for reproduction, and 8, 2, and 11 bug reports are removed due to failure to build the APK, environmental problems and cannot be reproduced, respectively. As shown in rows 1–6 of Table I, six bug reports from five applications are remained.

In ReCDroid, Zhao et al. [8] took the bug reports used in the FUSION [25] and YAKUSU [9] papers, and randomly crawled several bug reports from GitHub, to construct a benchmark consisted of 51 bug reports. According to the information provided by the ReCDroid paper, we download the specified versions of the Android application projects where the bugs are located as described in the bug reports, attempt to build and run the Android applications to reproduce the bugs in them. 9, 20, and 12 bug reports are removed, due to unavailable project source code, environmental problems, and cannot be reproduced, respectively. As shown in rows 7–16 of Table I, ten bug reports from ten applications are remained.

*4) Experimental Environment:* Based on the proposed method, a prototype tool is implemented on the basis of the PUMA framework to evaluate effectiveness. In the experiments, we compare CrPDroid with PUMA [14], APE [13], ReCDroid [8], and ReproBot [10]. PUMA and APE are commonly used Android automated testing tools and are used as baselines in comparative experiments of related studies, such as Humanoid [26] and ATUA [27]. The goal of ReCDroid and ReproBot is similar to CrPDroid, which is to reproduce the crashes described in the bug reports. It should be noted that the bug report analysis tools FUSION and YAKUSU are not chosen for this study. Although both FUSION and YAKUSU aim to generate event sequences for crash reproduction, their analysis results still rely on manual validation by developers, making them unsuitable for comparison with CrPDroid. The development and experiment environment of CrPDroid is a computer with 16-GB memory, a 6-core 3.3-GHz processor, Ubuntu 20.04, Android SDK (4.3-8.0), and JDK 1.8.0.

## B. Experimental Results

TABLE I
BASIC INFORMATION OF THE EXPERIMENTAL SUBJECTS

| Experimental Subject | Link to Bug Reports |
|---|---|
| Tomdroid | https://bugs.launchpad.net/tomdroid/+bug/1482559 |
| RadioBeacon (Bug1) | https://github.com/openbmap/radiocells-scanner-android/issues/239 |
| RadioBeacon (Bug2) | https://github.com/openbmap/radiocells-scanner-android/issues/173 |
| APhotoManager | https://github.com/k3b/APhotoManager/issues/175 |
| BetterBatteryStats | https://github.com/asksven/BetterBatteryStats/issues/871 |
| AnyMemo | https://github.com/helloworld1/AnyMemo/issues/502 |
| FastAdapter | https://github.com/mikepenz/FastAdapter/issues/394 |
| LibreNews-Android | https://github.com/milesmcc/LibreNews-Android/issues/22 |
| Transistor | https://github.com/y20k/transistor/issues/63 |
| Collect | https://github.com/getodk/collect/issues/2086 |
| Anki-Android | https://github.com/ankidroid/Anki-Android/issues/4586 |
| FlashCards | https://github.com/ASU-CodeDevils/FlashCards/issues/13 |
| Screenrecorder | https://github.com/vijai1996/screenrecorder/issues/32 |
| AIMSICD | https://github.com/CellularPrivacy/Android-IMSI-Catcher-Detector/issues/816 |
| microMathematics | https://github.com/mkulesh/microMathematics/issues/39 |
| AndroidDagger | https://github.com/vestrel00/android-dagger-butterknife-mvp/issues/46 |

*1) Results of RQ1 (What Impact Does the Fitness Calculation Parameter Have on the Ability of CrPDroid to Reproduce Crashes?):* To evaluate the impact of the parameter settings on the crash reproduction performance of CrPDroid, the parameter $K$ is set from 5 to 50, and the step size is 5 for CrPDroid. Then, the time required to reproduce the crash with different parameter settings is compared.

In Table II, the first column lists the 16 experimental subjects, and Columns 2–11 show the time CrPDroid takes to reproduce crashes with different $K$ values. As Table II shows, except for the crash in Screenrecorder, CrPDroid can reproduce the remaining 15 crashes when $K$ is set between 5 and 50. Further analysis of the crash in Screenrecorder indicates that reproducing this crash requires operations on the notification drawer of the Android device, which CrPDroid currently does not support. This limitation results in CrPDroid being unable to reproduce this crash for any value of $K$. Additionally, the reproduction times of CrPDroid tend to be stable as $K$ increases. For example, for Transistor and AndroidDagger, the minimum crash reproduction times of CrPDroid are 5.5 and 2.8 s, respectively, when the value of $K$ is 20; for Tomdroid, RadioBeacon (Bug1), and RadioBeacon (Bug2), the minimum crash reproduction times of CrPDroid are 11.4, 11.8, and 11.4 s, respectively, when the value of $K$ is 25; for APhotoManager and BetterBatteryStats, the minimum times required for CrPDroid to reproduce the crash are 12.6 and 21.7 s, respectively, when the value of $K$ is 30; for Collect and AIMSICD, the minimum times required for

TABLE II
TIME FOR CRPDROID TO REPRODUCE CRASHES UNDER DIFFERENT VALUES OF $K$ (IN SECONDS)

| Experimental subjects | $K$=5 | $K$=10 | $K$=15 | $K$=20 | $K$=25 | $K$=30 | $K$=35 | $K$=40 | $K$=45 | $K$=50 |
|---|---|---|---|---|---|---|---|---|---|---|
| Tomdroid | 54.0 | 49.0 | 31.3 | 13.9 | 11.4 | **11.3** | **11.3** | **11.3** | **11.3** | **11.3** |
| RadioBeacon (Bug1) | 147.8 | 132.1 | 14.4 | 12.4 | **11.8** | **11.8** | **11.8** | **11.8** | **11.8** | **11.8** |
| RadioBeacon (Bug2) | 110.9 | 106.0 | 17.8 | 16.7 | **14.4** | 14.5 | **14.4** | **14.4** | **14.4** | **14.4** |
| APhotoManager | 59.0 | 37.3 | 25.3 | 20.7 | 18.3 | **12.6** | **12.6** | **12.6** | **12.6** | **12.6** |
| BetterBatteryStats | 241.8 | 184.4 | 113.1 | 23.2 | 22.7 | **21.7** | **21.7** | **21.7** | **21.7** | **21.7** |
| AnyMemo | 1908.3 | 565.5 | 470.6 | 465.9 | 414.6 | 391.9 | 320.3 | 287.0 | **283.1** | 283.9 |
| FastAdapter | 120.7 | 43.1 | 27.9 | 16.5 | 13.2 | 7.9 | 7.1 | **6.4** | **6.4** | **6.4** |
| LibreNews-Android | 132.1 | 102.4 | 95.1 | 89.2 | 81.2 | 74.2 | 67.7 | 56.4 | **39.2** | **39.2** |
| Transistor | 10.2 | 6.8 | 6.6 | **5.5** | **5.5** | **5.5** | **5.5** | **5.5** | **5.5** | **5.5** |
| Collect | 15.2 | 13.3 | 13.2 | 12.6 | 9.2 | 8.7 | **8.2** | **8.2** | **8.2** | **8.2** |
| Anki-Android | 51.7 | 34.5 | 29.1 | 26.0 | 24.3 | 23.0 | 22.8 | 19.6 | **16.7** | **16.7** |
| FlashCards | 50.7 | 47.1 | 45.8 | 42.8 | 41.8 | 39.7 | 39.6 | **32.9** | **32.9** | **32.9** |
| Screenrecorder | T/O | T/O | T/O | T/O | T/O | T/O | T/O | T/O | T/O | T/O |
| AIMSICD | 14 | 11.5 | 10.3 | 10.2 | 10.2 | 9.2 | **8.6** | **8.6** | **8.6** | **8.6** |
| microMathematics | 113.2 | 113.2 | 100.3 | 90.5 | 89.0 | 87.0 | 64.3 | 64.2 | **61.5** | **61.5** |
| AndroidDagger | 4.0 | 3.9 | 3.5 | **2.8** | **2.8** | **2.8** | **2.8** | **2.8** | **2.8** | **2.8** |
| Average | 202.2 | 96.7 | 67.0 | 56.6 | 51.4 | 48.1 | 41.2 | 37.6 | **35.8** | **35.8** |

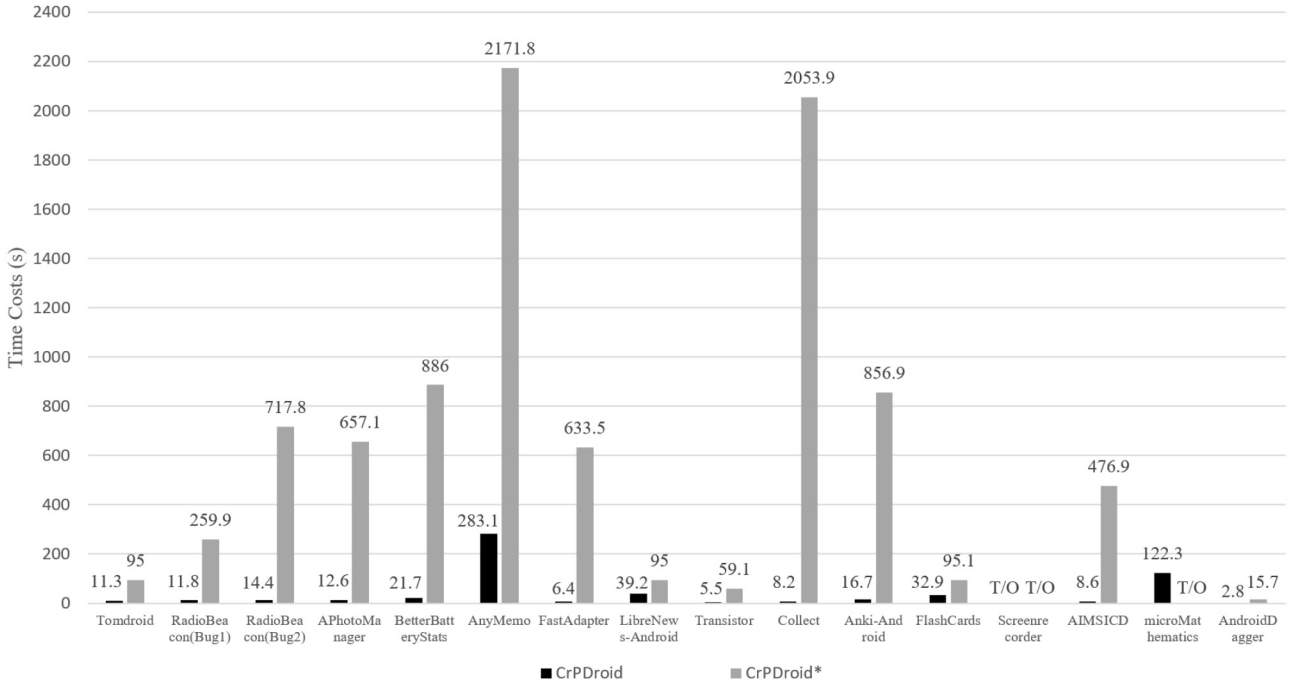


Fig. 5. Comparison between CrPDroid and CrPDroid* on crash reproduction.

CrPDroid to reproduce the crash are 8.2 and 8.6 s, respectively, when the value of $K$ is 35; for FastAdapter and FlashCards, the minimum times required for CrPDroid to reproduce the crash are 6.4 and 32.9 s, respectively, when the value of $K$ is 40; for AnyMemo, LibreNews-Android, Anki-Android, and microMathematics, the minimum times required for CrPDroid to reproduce the crash are 283.1, 39.2, 16.7, and 61.5 s, respectively, when $K$ is 45.

According to the experimental results, the crash reproduction performance of CrPDroid is affected by the value of $K$. When $K$ is 45, the crash described in the 15 bug reports can be reproduced by CrPDroid most efficiently. Therefore, $K$ is set to 45 in the following experiments for RQ2, RQ3, and RQ4.

*2) Results of RQ2 (How Does Widget Hierarchy Graph Guidance Affect the Performance of CrPDroid?):* For this RQ, CrPDroid is compared to CrPDroid*, which is not guided by widget hierarchy graphs. Without the guidance of the widget hierarchy graph, CrPDroid* cannot obtain the fitness values of widgets, and the fitness of all widgets in the application is set to $N$. At this time, the widgets have the same probability of being selected by CrPDroid* according to (3), and the exploration strategy of CrPDroid* degenerates to a random strategy.

Fig. 5 shows the time required for CrPDroid ($K = 45$) and CrPDroid* (CrPDroid without guidance by widget hierarchy graphs) to reproduce the crashes for the 16 experimental subjects. Due to the limitation of being unable to manipulate the notification bar, both CrPDroid and CrPDroid* cannot reproduce the crashes in Screenrecorder. For the other 15 experimental subjects, CrPDroid successfully reproduce all

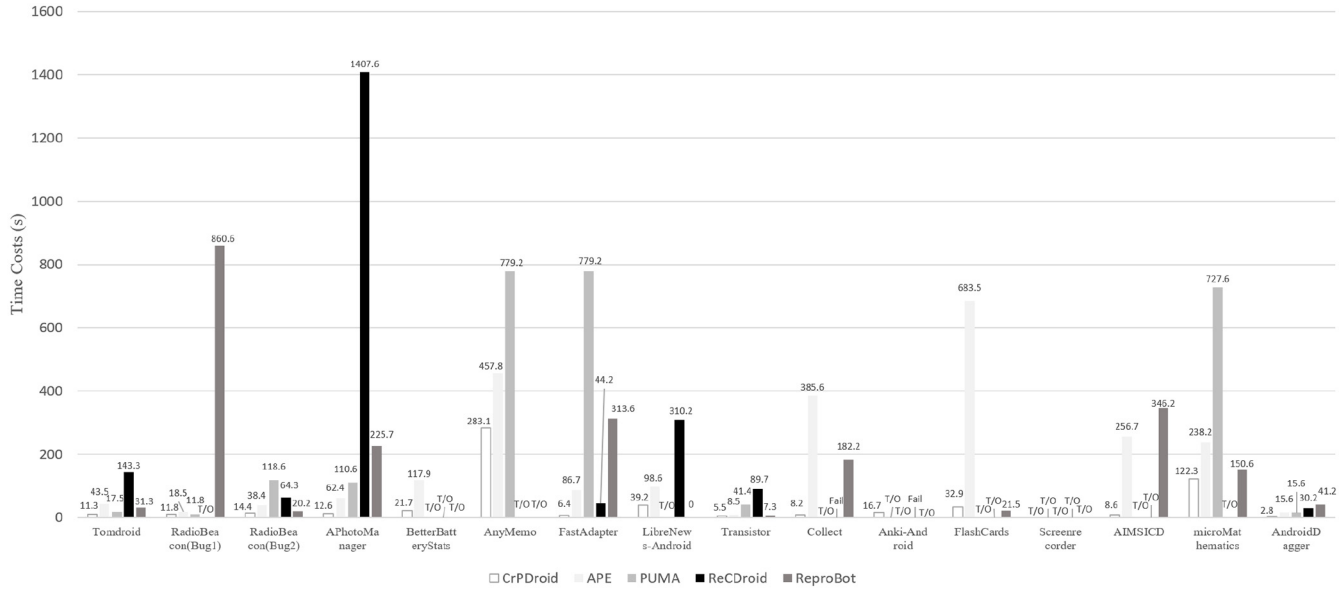


Fig. 6. Comparison of CrPDroid, APE, PUMA, and ReCDroid in terms of crash reproduction.

crashes, while CrPDroid* cannot reproduce the crash in microMathematics. For crashes that both CrPDroid and CrPDroid* could successfully reproduce, CrPDroid saves 97.20% of the time needed to reproduce crashes compared to CrPDroid* on average, because the reason is that CrPDroid* is not guided by the widget hierarchy graph and randomly allocates testing resources to the widgets of applications during exploration. The randomness results in CrPDroid* being unable to reproduce the crash in microMathematics, which has a large number of interfaces and widgets, within a limited time cost. In contrast, CrPDroid, which is guided by the widget hierarchy graph, prioritizes the allocation of testing resources to operations related to suspicious widgets. This makes CrPDroid more efficient than CrPDroid* in reproducing crashes.

The above experimental results show that, benefitting from a more efficient exploration strategy, CrPDroid and CrPDroid* can effectively reproduce most crashes. In addition, the widget hierarchy graph can further improve the efficiency of CrPDroid in reproducing crashes.

*3) Results of RQ3 (In Comparison With Other Automated Testing Tools for Android Applications, How Effectively and Efficiently Does CrPDroid Reproduce Crashes?):* This RQ compares CrPDroid, APE, PUMA, ReCDroid, and ReproBot for crash reproduction. The effectiveness of the tool is measured by the number of crashes successfully reproduced within a limited time and the time costs needed to reproduce crashes.

Fig. 6 shows the results of reproducing crashes with CrPDroid ($K = 45$), ReCDroid, ReproBot and the automated testing tools PUMA and APE. The Venn diagram of the crashes reproduced by different tools is shown in Fig. 7. Figs. 6 and 7 suggest that, due to the same limitation of being unable to operate the notification drawer, the crash in Screenrecorder that cannot be reproduced by CrPDroid also cannot be reproduced by ReproBot, PUMA and APE. For the remaining 15 crashes, the automated testing tool APE

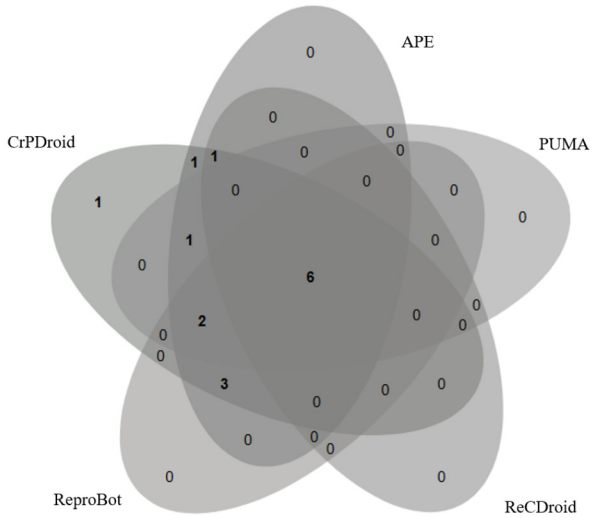


Fig. 7. Venn diagram of crashes reproduced by CrPDroid, APE, PUMA, ReCDroid, and ReproBot.

successfully reproduce 14 of them, except for the crash in Anki-Android. For 14 crashes that both CrPDroid and APE can reproduce, APE takes longer time than CrPDroid. CrPDroid reduced the average time to reproduce the 14 crashes by 76.87% compared to APE. Further analysis of the crash in Anki-Android, which APE fails to reproduce, suggests that APE expends a lot of testing resources on paths unrelated to reproducing the crash, making it unable to reproduce the crash within the limited time.

The automated testing tool PUMA can reproduce 10 out of 15 crashes, except for the crashes in LibreNews-Android, Collect, Anki-Android, FlashCards, and AIMSICD. Among the nine crashes that both CrPDroid and PUMA could reproduce, CrPDroid and PUMA took the same amount of time to reproduce the crash in RadioBeacon (Bug1), while CrPDroid

took less time to reproduce the other eight crashes. For crashes that both CrPDroid and PUMA could successfully reproduce, CrPDroid saved 81.94% of the time needed to reproduce crashes compared to PUMA on average. The reason is that PUMA's model-based exploration strategy does not consider suspicious widgets during the exploration process, thus resulting in lower efficiency in crash reproduction. When reproducing crashes in Collect, Anki-Android, and AIMSICD, this model-based exploration strategy allocates more testing resources to widgets unrelated to crash reproduction, making it unable to reproduce crashes within the limited time. With further analysis of the crash in BetterBatteryStats, LibreNews-Android, FlashCards, which cannot be reproduced by PUMA, we found that PUMA triggered an exception while backtracking the application, which caused PUMA to terminate and fail to reproduce the crash.

As a tool for reproducing crashes in bug reports, ReCDroid can only reproduce seven crashes and takes a longer time than CrPDroid. It should be noted that, for the 15 experimental subjects excluding Screenrecorder, CrPDroid fails to execute Collect and Anki-Android by using the scripts provided by ReCDroid. For the seven crashes that both CrPDroid and ReCDroid can reproduce, CrPDroid reduces time costs by 95.58% on average compared to ReCDroid. After analyzing the reasons for ReCDroid's poor performance in this experiment, it was found that ReCDroid aims to reproduce crashes from the textual descriptions in bug reports. The absence of adequate text describing the steps to reproduce the crash in bug reports can negatively affect its crash reproduction efficiency, as shown in the experiments on AnyMemo, FlashCards, and microMathematics. Second, ReCDroid does not support swipe gestures, which prevents it from reproducing crashes in RadioBeacon (bug1), BetterBatteryStats, and AIMSICD. Finally, ReCDroid dynamically builds and maintains a dynamic ordered event tree (DOET) during exploration to guide automated testing. Building and analyzing the DOET while exploring the application is time consuming. In contrast, the widget fitness used by CrPDroid to guide exploration is determined by the static analysis phase, without additional analysis during the exploration of the application.

The crash reproduction tool ReproBot, similar to ReCDroid, aims to reproduce crashes with the textual descriptions in bug reports. ReproBot can reproduce 11 out of 15 crashes. However, for these 11 reproduced crashes, ReproBot takes longer time than CrPDroid. CrPDroid reduces the average time by 76.55% compared to ReproBot. This is because the crash reproduction steps included in the bug reports are often incomplete, and ReproBot consumes more time to explore the application to find the missing crash reproduction steps. Moreover, the matching analysis of crash reproduction steps and UI events of the application interface is a time-consuming task. In contrast, CrPDroid obtains a widget hierarchy graph during the static analysis phase, that contains the relationships between widgets and interfaces, as well as the interface jumps. CrPDroid can quickly locate suspicious widgets with this graph. Further analysis of the four crashes that ReproBot fails to reproduce, suggests that for Anki-Android and AnyMemo, ReproBot consumes a lot of time to explore the application to find the missing crash reproduction steps, making it unable to reproduce the crash within the limited time. For LibreNews-Android and BetterBatteryStats, ReproBot cannot accurately match the crash reproduction steps with the UI events of the application during the exploration, resulting in unable to reproduce the crashes.

As the experimental results show, CrPDroid outperforms APE, PUMA, ReCDroid, and ReproBot in terms of the number of crashes reproduced. In addition, CrPDroid is more efficient than PUMA, APE, ReCDroid, and ReproBot in reproducing application crashes.

*4) Results of RQ4 (Can CrPDroid Concentrate More Testing Resources on Suspicious Widgets Than Other Automated Testing Tools for Android Applications?):* For this RQ, CrPDroid, APE, PUMA, ReCDroid, and ReproBot are compared in terms of the testing resources allocated to suspicious widgets. In some cases, PUMA may terminate the exploration earlier than 30 min if it is determined that the application has been fully explored or that no more interfaces can be explored. In this case, PUMA will be restarted so that the exploration time accumulates for 30 min and the operations on widgets for each exploration accumulate. For this RQ, the number of operations to suspicious widgets is used to evaluate the testing resource allocation of each tool on suspicious widgets.

Table III shows how CrPDroid, APE, PUMA, and ReCDroid operate on suspicious widgets in the same period of time. Columns 2, 6, 10, 14, and 18 are the times elapsed from the start of the experiment to the first operation on the suspect widget. $Exe_s$, in Columns 3, 7, 11, 15, and 19, represents the total number of operations on suspicious widgets within 30 min. $Exe_a$, in Columns 4, 8, 12, 16, and 20, represents the total number of operations on all the widgets within 30 min. $Exe_s/Exe_a$, in Columns 5, 9, 13, 17, and 21, represents the ratio of operations on suspicious widgets to the number of operations on all widgets.

The results show that in the same period of testing time, compared with PUMA, the total operations of CrPDroid on all widgets increases by 25.69% on average, the total operations of CrPDroid on suspicious widgets increases by 87.57% on average, and the average $Exe_s/Exe_a$ value of CrPDroid is 35.18%, which is 6.2 times greater than PUMA. Compared with APE, the total operations of CrPDroid on all widgets is fewer, but the operations of CrPDroid on suspicious widgets increases by 44.07% on average, and the average $Exe_s/Exe_a$ value is 7.6 times greater than APE. Compared with ReCDroid, the total operations of CrPDroid on all the widgets increases by 65.02% on average, the operations of CrPDroid on suspicious widgets increases by 88.70% on average, and the average $Exe_s/Exe_a$ value of CrPDroid is 3.2 times greater than ReCDroid. Compared with ReproBot, the total operations of CrPDroid on all widgets increases by 14.23% on average, the operations of CrPDroid on suspicious widgets increases by 68.93% on average, and the average $Exe_s/Exe_a$ value of CrPDroid is 2.8 times greater than ReproBot.

In addition, in terms of the time cost from the beginning of the experiment to the first operation on a suspicious widget,

TABLE III
COMPARISON OF TESTING RESOURCE ALLOCATION

| Experimental subjects | CrPDroid | | | | APE | | | | PUMA | | | | ReCDroid | | | | ReproBot | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $T_f$[1] | $Exe_s$ | $Exe_a$ | $Exe_s/Exe_a$ | $T_f$ | $Exe_s$ | $Exe_a$ | $Exe_s/Exe_a$ | $T_f$ | $Exe_s$ | $Exe_a$ | $Exe_s/Exe_a$ | $T_f$ | $Exe_s$ | $Exe_a$ | $Exe_s/Exe_a$ | $T_f$ | $Exe_s$ | $Exe_a$ | $Exe_s/Exe_a$ |
| Tomdroid | **2.8** | **193** | 521 | **37.04%** | 278.2 | 34 | **2039** | 1.67% | 9.7 | 3 | 438 | 0.68% | 143.3 | 15 | 128 | 11.72% | 31.3 | 157 | 485 | 32.37% |
| RadioBeacon (Bug1) | **2.8** | **203** | 510 | **39.80%** | 114.6 | 41 | **1960** | 2.09% | 2.8 | 5 | 310 | 1.61% | T/O | 0 | 158 | 0% | 31.2 | 103 | 425 | 24.24% |
| RadioBeacon (Bug2) | **5.8** | **142** | 542 | **26.20%** | 6.0 | 8 | **2106** | 0.38% | 8.9 | 45 | 345 | 13.04% | 64.3 | 32 | 190 | 16.84% | 20.2 | 75 | 365 | 20.55% |
| APhotoManager | **6.5** | **167** | 482 | **34.65%** | 106.5 | 65 | **1492** | 4.36% | 104.4 | 3 | 381 | 0.79% | 231.5 | 7 | 143 | 4.90% | 129.2 | 46 | 497 | 9.26% |
| BetterBatteryStats | **11.3** | **143** | 377 | **37.93%** | 12.3 | 16 | **2120** | 0.75% | 100.8 | 12 | 336 | 3.57% | T/O | 0 | 87 | 0% | 428.3 | 72 | 461 | 15.62% |
| AnyMemo | **2.7** | **159** | 418 | **38.04%** | 91.6 | 8 | **2314** | 0.35% | 19.2 | 24 | 348 | 6.90% | 632.6 | 16 | 113 | 14.16% | 124.5 | 16 | 390 | 4.10% |
| FastAdapter | **3.2** | **175** | 506 | **34.58%** | 7.5 | 87 | **1950** | 4.46% | 5.7 | 23 | 257 | 8.94% | 40.2 | 43 | 164 | 26.22% | 92.1 | 12 | 418 | 2.87% |
| LibreNews-Android | **5.3** | 151 | 480 | **31.46%** | 13.3 | **167** | **2846** | 5.87% | 15.6 | 4 | 413 | 0.97% | 115.3 | 57 | 206 | 27.67% | 16.6 | 106 | 446 | 23.77% |
| Transistor | **2.6** | 192 | 661 | **29.05%** | 4.2 | **407** | **1882** | 21.63% | 8.5 | 3 | 335 | 0.90% | 45.3 | 32 | 181 | 17.68% | 3.4 | 66 | 397 | 16.62% |
| Collect | **5.7** | **158** | 598 | **26.42%** | 297.1 | 8 | **2505** | 0.32% | T/O | 0 | 420 | 0% | - | - | - | - | 58.2 | 47 | 467 | 10.06% |
| Anki-Android | **10.5** | **287** | 508 | **56.50%** | 96.7 | 17 | **2178** | 0.78% | 2.7 | 32 | 371 | 8.63% | - | - | - | - | 122.2 | 57 | 431 | 13.23% |
| FlashCards | **13.1** | **126** | 396 | **31.82%** | 631.2 | 4 | **1931** | 0.21% | 16.2 | 57 | 462 | 12.34% | T/O | 0 | 210 | 0% | 18.7 | 41 | 331 | 12.39% |
| Screenrecorder | **2.6** | **150** | 447 | **33.56%** | 87.5 | 147 | **2544** | 5.78% | 2.6 | 2 | 372 | 0.54% | 492.2 | 4 | 297 | 1.35% | 484.2 | 5 | 483 | 1.04% |
| AIMSICD | **6** | **156** | 497 | **31.39%** | 51.6 | 15 | **2385** | 0.63% | 161.5 | 22 | 282 | 7.80% | 1014.7 | 17 | 186 | 9.14% | 327.1 | 12 | 487 | 2.46% |
| microMathematics | **7.7** | **117** | 576 | **20.31%** | 233.6 | 12 | **1745** | 0.69% | 54.7 | 12 | 471 | 2.55% | T/O | 0 | 177 | 0% | 139.3 | 26 | 492 | 5.28% |
| AndroidDagger | **2.8** | 312 | 577 | **54.07%** | 15.6 | **546** | **2242** | 24.35% | 15.6 | 102 | 473 | 21.56% | 30.2 | 61 | 240 | 25.42% | 41.2 | 32 | 366 | 8.74% |
| Average | **5.7** | **177** | 506 | **35.18%** | 128.0 | 99 | **2140** | 4.65% | 145.6 | 22 | 376 | 5.68% | 715.0 | 20 | 177 | 11.08% | 129.2 | 55 | 434 | 12.7% |

[1] The time unit of $T_f$ is seconds.

the average time of CrPDroid is 95.55%, 96.09%, 99.20%, and 95.59% less than that of APE, PUMA, ReCDroid and ReproBot, respectively. It should be noted that CrPDroid is implemented based on PUMA, but the total number of operations of PUMA on all the widgets is less than that of CrPDroid in the same period of testing time. One reason is that PUMA restarts the application when it finishes exploring a path, but restarting the application takes time.

To summarize, CrPDroid is more efficient in reproducing crashes, prioritizing testing resources for suspicious widgets, and performing more operations on suspicious widgets. As a result, CrPDroid reproduces crashes faster than PUMA, APE, ReCDroid, and ReproBot.

## IV. DISCUSSION

### A. Widget Hierarchy Graphs Analysis

The widget hierarchy graph, which describes the relationships between interfaces and widgets, as well as the interface jumps, is used by CrPDroid to guide automated testing. To further evaluate the accuracy of the widget hierarchy graph, we check 5 of the graphs statically constructed by CrPDroid for 16 experimental subjects, namely, APhotoManager, LibreNews-Android, Transistor, microMathematics, and AndroidDagger. The graphs constructed for the five experimental subjects are manually compared with the actual Android applications. The results show that the widget hierarchy graphs constructed by CrPDroid for APhotoManager, Transistor, and AndroidDagger are accurate, and CrPDroid can quickly reproduce crashes by using the widget hierarchy graphs. However, the widget hierarchy graphs constructed by CrPDroid for LibreNews-Android and microMathematics are incomplete. Specifically, 1 out of 4 and 3 widgets required to reproduce the crash of LibreNews-Android and microMathematics are not contained in the widget hierarchy graphs of the two application, respectively. Considering this situation, we set the fitness of widgets not identified during the static analysis phase to a fixed value *N* to avoid the case in which some widgets are never selected for operation during the exploration phase because they were not identified. This strategy enables CrPDroid to reproduce crashes by attempting to explore unknown paths when it fails to reproduce a crash based on the widget hierarchy graph, although this consumes some additional time costs.

### B. Crash Reproduction Steps Analysis

Although CrPDroid aims to reproduce the crashes described in the bug reports, it does not focus on strictly following the steps in the bug reports to reproduce the crashes. We conducted a comparative analysis between the steps of crashes successfully reproduced by CrPDroid and the steps in the bug reports to find out their relationships. The results show that when the crash reproduction steps in the bug reports are complete, the actual crash reproduction steps of CrPDroid match them. This is because CrPDroid treats the widget operated in each crash reproduction step of the bug reports as suspicious widgets, and prioritizes the testing resources to the suspicious widgets to reproduce the crash by operating the suspicious widgets. However, when the crash reproduction steps provided in the bug reports are incomplete, the actual crash reproduction steps generated by CrPDroid differ from them. Specifically, the actual crash reproduction steps are more like a complement for the steps described in the bug reports. Although there could be some redundant operation steps. This is because when CrPDroid cannot reproduce the crash using the information in the bug reports, it attempts to explore alternative paths within the application to assist in crash reproduction.

### C. Limitations

CrPDroid relies on stack trace and widget information in bug reports to reproduce crashes and cannot obtain assistance from information provided in other forms, such as attachments or screenshots. If the stack trace and widget information provided in the bug report is insufficient, it will reduce the reproduction efficiency of CrPDroid. In extreme cases,

CrPDroid may be used as an automated testing tool with a random strategy (i.e., CrPDroid* in the experiment). For instance, in the AnyMemo application, which is one of the experimental subjects, the bug report primarily consists of attachments and screenshots. CrPDroid cannot extract useful information, resulting in a significantly longer crash reproduction time than other experimental subjects (as shown in Fig. 6). In subsequent improvements, it is possible to consider further utilizing the information provided in other forms in bug reports. For example, widget information can be extracted from screenshots using computer vision techniques to guide automated testing.

In addition, the efficiency of CrPDroid in executing applications is low. In our experiments, CrPDroid performs significantly fewer operations than APE within the same testing period, as shown in Table III. One significant reason is that CrPDroid waits for a fixed time after performing an operation to ensure its completion, as some operations may generate animation effects. Subsequent improvements can be made for the waiting strategy. For example, CrPDroid could monitor whether the interface has finished loading after operating a widget, and immediately operate the next widget after the loading is completed.

### D. Threats to Validity

*1) External Validity:* External validity threats arise from the representativeness of the selected evaluation subjects and bug reports on the one hand and the generality of CrPDroid on the other. To ensure the representativeness of the evaluation subjects and bug reports, the Android applications selected in the experiment are widely used in related testing works [21], [22], [23], and the source codes are all open source on GitHub or F-Droid. Bug reports are obtained from the respective bug tracking system or comment section of the application. To improve the generality of CrPDroid, we implement CrPDroid based on PUMA. PUMA is a programmable Android testing framework that supports user-defined exploration strategies and collects dynamic information. PUMA has been validated on 3600 apps in Google Play [14] and is used as the base framework in the related works of Liu et al. [28] and Jiang et al. [29].

*2) Internal Validity:* The internal validity threat mainly comes from the accuracy of the constructed widget hierarchy graph and the correctness of the exploration of the application under test. To improve the accuracy of the widget hierarchy graph, we use the static analysis tool FlowDroid to analyze and obtain function calls from the APK file of the application under test. FlowDroid is widely used for data flow analysis of Android applications and Java programs. The PUMA framework is used to ensure that the exploration of applications under test is accurate for CrPDroid. PUMA has been widely used in analyzing program attributes (such as application state and widget information) [14]. Moreover, we check and test the written code for constructing widget hierarchy graphs and calculating the fitness of widgets to minimize the risk of invalidity.

## V. Related Work

This section introduces related work on Android GUI testing and bug report analysis.

### A. Android GUI Testing

GUI testing has been extensively used to detect application crashes and achieve high code coverage. Automated testing techniques can explore GUI events of the application under test by using different strategies (such as random, model-based, and reinforcement learning-based exploration strategies).

Some related work uses a random exploration strategy, generating pseudorandom events to perform fuzz testing on the application under test. Monkey [7] is the most commonly used random strategy-based automated testing tool, which generates a pseudorandom stream of GUI events by randomly interacting with screen coordinates. The random strategy used by Monkey performs well on some benchmark applications. However, the test cases generated by Monkey contain many invalid or redundant events, which threatens the efficiency of the testing process. Dynodroid [30] improves the exploration method of randomly interacting with screen coordinates. Its exploration of the application is a cycle of observation, selection, and execution. In the observation phase, Dynodroid determines the layout of the widgets on the current screen and the expected input type for each widget. In the selection phase, Dynodroid tends to select widgets that are not frequently selected. Finally, in the execution phase, Dynodroid operates on the selected widget. In addition, Dynodroid allows users to pause automatic exploration and resume exploration after manual operations (such as authentication and password input) by the user.

There is also some work that generates test cases based on application models. Taking GUIRipper [31] as an example, it implements a model-based exploration strategy, which dynamically builds a model of an application while exploring the application with the DFS strategy. The application under test will be re-explored if GUIRipper judges that any new interface cannot be reached. Muangsiri and Takada [32] also proposed a model-based exploration method, a behavioral-based GUI testing method for mobile applications that achieves high code coverage. The method creates a behavioral model from usage logs by applying a statistical model. The events within the behavioral model are mapped to GUI components in a GUI tree. During testing, the method updates the model dynamically to increase the probability of an event that rarely or never occurs when users use the application. ATUA [27] uses the application model to guide the dynamic exploration of the application under test, thereby achieving significantly higher coverage of the code affected by updates with a much smaller number of test inputs. The application model is initially created by static program analysis procedures and then refined during testing.

Some work uses reinforcement learning techniques to guide Android application testing. For example, Q-testing [21] applies reinforcement learning techniques to test Android applications with a curiosity-driven exploration strategy. During the exploration process, Q-testing records and maintains a set of application states that have been visited and

then calculates reinforcement learning rewards based on the difference between the current state and the recorded states. By changing the importance of the application state under test, Q-testing can dynamically adjust rewards for specific events. QBE [33] also uses reinforcement learning techniques to explore the application under test, implementing a fully automated black-box testing method. QBE uses reinforcement learning techniques to generate a Q-Matrix to evaluate the probability of operating each widget. During the exploration of the application under test, the widget to be operated is selected from the application interface based on the Q-Matrix. Romdhana et al. [34] first proposed a deep reinforcement learning method called ARES for automatic black-box testing of Android applications. Deep reinforcement learning is a recent extension of reinforcement learning that takes advantage of the learning capabilities of neural networks. Due to this deep neural network, ARES achieves high scalability, general applicability, and the ability to handle complex application behavior.

The above techniques have successfully detected application bugs or achieved high code coverage. However, these tools are not designed to make use of bug reports, and their exploration scope includes all reachable paths of the application under test, resulting in less efficiency in reproducing crashes.

### B. Bug Report Analysis

Some related studies have focused on the importance of bug reports in the quality assurance of Android applications.

FUSION [25] assists users in automatically generating operation steps for reproducing bug reports by dynamically analyzing the GUI events of Android applications. With FUSION, users can create more comprehensive and accurate bug reports, and developers can obtain operable information from bug reports, which can help reproduce and fix Android application bugs. Unlike our method, FUSION focuses only on automatically generating bug reports and cannot reproduce application crashes based on bug reports.

In addition to generating bug reports for Android applications, YAKUSU [9] extracts abstract operation steps describing how to reproduce bugs by combining static analysis and natural language processing (NLP). The abstract operation steps are used to guide the dynamic search process, which will find a sequence of GUI operations that match the abstract operation steps. YAKUSU aims to help developers analyze bug reports and generate test cases based on those reports, rather than reproducing the crashes described in the bug reports. Therefore, the event sequences generated by YAKUSU cannot guarantee the reproduction of the crashes described in the bug reports and still require manual inspection by developers. In contrast, our work automatically reproduces the crashes described in the bug reports.

Along with directly using bug reports, improving the quality of bug reports is also important. Chaparro et al. [35] developed three versions of DEMIBUD based on regular expressions, heuristics, NLP, and machine learning (ML) to detect missing information in bug reports. DEMIBUD can remind submitters to fill in missing content when writing bug reports and help developers evaluate the quality of bug reports. In addition, DEMIBUD can be used to expand existing bug report quality models. Unlike CrPDroid, DEMIBUD aims to improve the quality of bug reports and cannot reproduce crashes using bug reports.

ReCDroid [8], ReproBot [10], and ScopeDroid [11] are similar to CrPDroid, which focused on reproduce crashes in bug reports. ReCDroid [8] reproduces crashes in two stages: 1) bug report analysis and 2) dynamic exploration. In the bug report analysis stage, ReCDroid uses NLP techniques to extract GUI event representations for each step from the bug report. In the dynamic exploration stage, ReCDroid explores the application under test based on the GUI event representations obtained in the previous stage. During exploration, ReCDroid builds and maintains a DOET, searches for and fills in missing steps in the bug report, and finally outputs the event sequence to a script.

ReproBot [10] conducts crash reproduction in two phases. In the first phase, it analyzed natural language sentences describing crash reproduction steps to extract information about the widgets to be operated. In the second stage, ReproBot explored the application by operating widgets based on the extracted widget information. During the exploration, $Q$-learning was used by ReproBot to find the missing steps required for the crash reproduction.

ScopeDroid [11] reproduced crashes in three phases. First, ScopeDroid used an automated testing tool to explore the application under test and constructed its initial state transition graph. Then it designed a multimodal neural matching network to derive the fuzzy matching matrix between all candidate GUI events and reproducing steps. Finally, ScopeDroid planed the crash reproduction path with the initial state transition graph and the fuzzy matching matrix to guide the exploration accordingly. The path guided the exploration, and the state transition graph was updated during the exploration.

## VI. Conclusion

In this article, we present CrPDroid, a fast crash reproduction method for Android applications based on widget hierarchy graphs. By automatically analyzing the project file of the application under test, CrPDroid creates a widget hierarchy graph, which is used in combination with the bug report to generate test scripts for reproducing crashes. The experimental results show that CrPDroid is able to reproduce the crashes described in the bug reports and outperforms PUMA, APE, and ReCDroid in terms of time costs.

In the future, we intend to use information retrieval-based bug localization techniques to automatically find suspicious widgets and further improve the efficiency of CrPDroid in reproducing crashes.

## References

[1] L. Chettri and R. Bera, "A comprehensive survey on Internet of Things (IoT) toward 5G wireless systems," *IEEE Internet Things J.*, vol. 7, no. 1, pp. 16–32, Jan. 2020.

[2] T. Lei, Z. Qin, Z. Wang, Q. Li, and D. Ye, "EveDroid: Event-aware android malware detection against model degrading for IoT devices," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6668–6680, Aug. 2019.

[3] J. Wang et al., "IoT-praetor: Undesired behaviors detection for IoT devices," *IEEE Internet Things J.*, vol. 8, no. 2, pp. 927–940, Jan. 2021.

[4] "Number of available applications in the Google play store from december 2009 to march 2023." Statista. 2023. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store

[5] "88% of people will abandon an app because of bugs." Applause. 2017. [Online]. Available: https://www.applause.com/blog/app-abandonment-bug-testing

[6] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Proc. IEEE Int. Conf. Softw. Test., Verif. Valid. (ICST)*2016, pp. 33–44.

[7] "UI/application exerciser monkey," Android.com. 2021. [Online]. Available: https://developer.android.com/studio/test/monkey

[8] Y. Zhao et al., "ReCDroid: Automatically reproducing android application crashes from bug reports," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 128–139.

[9] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2018, pp. 141–152.

[10] Z. Zhang, R. Winn, Y. Zhao, T. Yu, and W. G. Halfond, "Automatically reproducing android bug reports using natural language processing and reinforcement learning," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2023, pp. 411–422.

[11] Y. Huang et al., "Context-aware bug reproduction for mobile apps," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2336–2348.

[12] G. Lin, Z. Zhang, and Z. Cui, "Widget hierarchy graph guided crash reproduction method for android applications," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, 2023, pp. 584–587.

[13] T. Gu et al., "Practical GUI testing of android applications via model abstraction and refinement," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 269–280.

[14] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. Annu. Int. Conf. Mobile Syst., Appl., Services*, 2014, pp. 204–217.

[15] P.-W. LI, Y.-Q. Jiang, F.-Y. Xue, J.-J. Huang, and C. Xu, "A robust approach for android malware detection based on deep learning," *Acta Electronica Sinica*, vol. 48, no. 8, pp. 1502–1508, 2020.

[16] "The activity lifecycle." Android.com. 2022. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle

[17] M. Honnibal and I. Montani, "spaCy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing," *To Appear*, vol. 7, no. 1, pp. 411–420, 2017.

[18] A. Kao and S. Poteet, *Natural Language Processing and Text Mining*. London, U.K.: Springer, 2007.

[19] "Word2Vec." Github.com. 2021. [Online]. Available: https://github.com/dav/word2vec

[20] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Proc. IEEE Work. Conf. Min. Softw. Repos. (MSR 2010)*, 2010, pp. 118–121.

[21] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2020, pp. 153–164.

[22] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2015, pp. 429–440.

[23] A. Sadeghi, R. Jabbarvand, and S. Malek, "paTDroid: Permission-aware GUI testing of android," in *Proc. Joint Meet. Found. Softw. Eng.*, 2017, pp. 220–232.

[24] "Open-source android apps." Github.com. 2022. [Online]. Available: https://github.com/pcqpcq/open-source-android-apps

[25] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *Proc. Joint Meet. Found. Softw. Eng.*, 2015, pp. 673–686.

[26] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, 2019, pp. 1070–1073.

[27] C. D. Ngo, F. Pastore, and L. Briand, "Automated, cost-effective, and update-driven app testing," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 31, no. 4, pp. 1–51, 2022.

[28] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proc. Annu. Int. Conf. Mobile Syst., Appl., Services*, 2015, pp. 89–103.

[29] B. Jiang, Y. Zhang, W. K. Chan, and Z. Zhang, "A systematic study on factors impacting GUI traversal-based test case generation techniques for android applications," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 913–926, Sep. 2019.

[30] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proc. Joint Meet. Found. Softw. Eng.*, 2013, pp. 224–234.

[31] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 258–261.

[32] W. Muangsiri and S. Takada, "Random GUI testing of android application using behavioral model," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 09n10, pp. 1603–1612, 2017.

[33] Y. Koroglu et al., "QBE: QLearning-based exploration of android applications," in *Proc. IEEE Int. Conf. Softw. Test., Verif. Valid. (ICST)*, 2018, pp. 105–115.

[34] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 31, no. 4, pp. 1–29, 2022.

[35] O. Chaparro et al., "Detecting missing information in bug descriptions," in *Proc. Joint Meet. Found. Softw. Eng.*, 2017, pp. 396–407.

**Zhanqi Cui** (Member, IEEE) received the B.Eng. degree from the Software Institute, Nanjing University, Nanjing, China, in 2005, and the Ph.D. degree from the Department of Computer Science and Technology, Nanjing University in 2011.

He is a Professor with the Computer School, Beijing Information Science and Technology University, Beijing, China. His main research interests include intelligent software analysis and testing technology.

**Gaoyi Lin** is currently pursuing the master's degree with the Computer School, Beijing Information Science and Technology University, Beijing, China.

His main research interests include intelligent software analysis and testing technology.

**Liwei Zheng** received the B.Eng. degree and the M.S. degree from the Department of Computer Science, Taiyuan University of Technology, Taiyuan, China, in 2001 and 2004, respectively, and the Ph.D. degree from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China, in 2009.

He is an Associate Professor with the Computer School, Beijing Information Science and Technology University, Beijing. His main research interests include requirement engineering and trusted computing.

**Zhihua Zhang** received the B.Eng. degree in computer and application and the M.S. degree in computer application from Harbin University of Science and Technology, Harbin, China, in 1993 and 1996, respectively.

She is an Associate Professor with the Computer School, Beijing Information Science and Technology University, Beijing, China. Her main research interests are in software testing.