Widget Hierarchy Graph Guided Crash Reproduction Method for Android Applications

Gaoyi Lin, Zhihua Zhang, Zhanqi Cui*
Computer School, Beijing Information Science and Technology University, Beijing, China
Email: {lingaoyi1998, zhang_zh, czq}@bistu.edu.cn

***Abstract*—To improve the efficiency of fixing bugs, mobile application developers must reproduce bugs reported by testers or users as quickly as possible. Automated testing tools can help but are not designed for reproducing bug reports. To improve the efficiency of reproducing crashes, we propose a widget hierarchy graph guided crash reproduction method for Android apps. It builds a widget hierarchy graph, locates suspicious widgets using bug reports and project files, calculates widget fitness, and guides automated testing to reproduce crashes quickly. To evaluate the effectiveness of the proposed method, experiments are conducted on real Android application bug reports and compared with the automated testing tools APE and PUMA. Experimental results show that our method successfully reproduces six bug reports that cause Android app crashes. In addition, compared with APE and PUMA, the average time for our method to reproduce crashes decreased by 51.94% and 71.47%.**

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

I. Introduction

As of Dec 2022, Android accounted for 72.37% of the mobile operating system market [1]. With its growing popularity, Android applications have proliferated. Developers face fierce competition and must release new versions of applications quickly, making thorough testing difficult. This results in bugs in released versions, increasing testing and maintenance costs and challenging app robustness and reliability.

Bugs in mobile applications result in user loss [2], so developers must quickly respond and fix them, especially application crashes that directly affect usability [3]. Reproducing crashes may require complex operations and can be inefficient when done manually. Automated testing tools can trigger some crashes but lack pertinence and efficiency. Software projects use bug tracking systems (such as Bugzilla[1], GitHub Issue Tracker[2], etc.) to manage testing and accelerate bug fixing. When reporting bugs, users and testers usually provide application versions, system versions, bug screenshots, stack traces, widget information, etc. Reviewing bug reports is an important way to find and reproduce crashes. By using information from bug reports, a developer can refine the search scope and allocates more testing resources to the locations related to the bug reports to improve crash reproduction efficiency. However, existing automated testing tools rarely pay attention to bug reports.

*Zhanqi Cui is the corresponding author.

[1]Bugzilla. https://bugzilla.mozilla.org/describebekeywords.cgi

[2]GitHub Issue Tracker. https://github.com

DOI reference number: 10.18293/SEKE2023-066

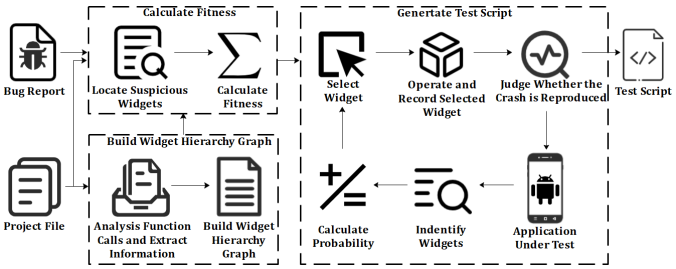

Fig. 1: The Framework of Widget Hierarchy Graph Guided Crash Reproduction Method for Android Applications.

It is possible to increase the efficiency of reproducing a crash by using the bug report information effectively and concentrating more testing resources on suspicious widgets. Therefore, we propose a widget hierarchy graph guided crash reproduction method for Android applications. By automatically analyzing the project file of the application under test, it creates a widget hierarchy graph, which is used in combination with the bug report to generate test scripts for reproducing crashes. To evaluate the effectiveness of our method, we implement the prototype tool based on this method and conduct experiments on 6 bug reports of 5 Android applications. We compare it with advanced automated testing tools APE [4] and PUMA [5]. The experimental results show that our method and APE both reproduced the application crashes mentioned in the 6 bug reports, while PUMA only reproduced 5. Compared with PUMA and APE, our method saves 71.47%, 51.94% time to reproduce the crashes in average, respectively.

This paper makes following contributions.

- proposes a widget hierarchy graph guided crash reproduction method for Android applications, to improve the efficiency of reproducing crashes by using widget hierarchy graphs;
- experiments are carried on a set of real applications to evaluate the effectiveness of the proposed method.

II. Widget Hierarchy Graph Guided Crash Reproduction Method for Android Applications

Fig. 1 shows the framework of the widget hierarchy graph guided crash reproduction method for Android applications. It consists of three components: (1) Build Widget Hierarchy Graph, (2) Calculate Fitness of Widgets, (3) Generate Test Script. These components are discussed in the following sections.

A. Build Widget Hierarchy Graph

We use a widget hierarchy graph to describe the relationships between widgets in an 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:

- $V = \{v_1, v_2, \dots, v_i, \dots\}$ is the set of nodes in the graph, and the node $v_i \in V$ is a function in A;
- $C = \{c_{1,2}, c_{3,4}, \dots, c_{k,l}, \dots\}$ and $J = \{j_{1,2}, j_{3,4}, \dots, j_{m,n}, \dots\}$ are the 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 a interface jump in A;
- W is the set of widgets in A;
- P is the 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 $Annotation(v_i) = \emptyset$, means that 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 in A can call the function v_l. For the interface jump $j_{m,n} \in J$, $From(j_{m,n}) = p_m \in P$, $To(j_{m,n}) = p_n \in P$, indicating that the interface p_m in A can jump to the 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, we analyze the APK file of the application under test for function calls. For the Android applications A, the set of functions V and the set of function calls C can be obtained directly from the function call. FlowDroid[3] is used to get function calls. To simplify the analysis of Android applications, FlowDroid generates dummy methods that represent the order of implicit invocations of lifecycle callback methods and GUI callback methods in applications.

After getting the set of functions V and the set of function calls C, we analyze 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 directly from the project file of the application under test. Then, analyze the project file to obtain the method related to the set of widgets W (when operating one widget, if the widget automatically calls one method, it is said that the widget is related to the method). 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, traverse the set of functions V, if the function v_u is a dummy function, 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 where W_u is located to the interface set P. Finally, if the interface p_m can jump to the interface p_n, add the interface jump $j_{m,n}$ to J, where $From(j_{m,n}) = p_m$, $To(j_{m,n}) = p_n$.

[3]FlowDroid. https://github.com/secure-software-engineering/FlowDroid

B. Calculate Fitness of Widgets

Our method guides the automated testing by using the fitness of each widget. The fitness of widgets can be obtained through widget hierarchy graph and suspicious widgets. To get the set of suspicious widgets W_{susp}, it uses the information of widgets or exception trace stack contained in the bug report to find the widgets related to the bug report from the project file of the application under test.

Algorithm 1 Calculate Fitness of Widgets for Application

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

```
1: F ← ∅
2: P_locate ← FindPageOfSuspWidget(W_susp, G)
3: for each page p_n ∈ P_locate do
4:     for each widget w_u^n ∈ p_n do
5:         F.add(w_u^n, CalFitWithFormula1(w_u^n, W_susp))
6: while first cycle or p_start ≠ ∅ do
7:     p_start ← FindStartingPageOfJump(P_locate, G)
8:     if P_start ≠ ∅ then
9:         for each page p_m ∈ P_start do
10:            for each widget w_v^m ∈ p_m do
11:                F.add(w_v^m, CalFitWithFormula2(w_v^m, G))
12:            P_locate = P_start
13: return F
```

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

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

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

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

For the interface $p_m \in P_{start}$ (line 7), there are $p_n \in P_{locate}$ and $j_{m,n} \in J$, that $From(j_{m,n}) = p_m$ and $To(j_{m,n}) = p_n$. We uses the formula (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. In order to make such widgets more likely to be covered during testing, for the $w_v^m \in p_m$, if w_v^m is the widget in p_m that can trigger the interface jump to p_n, its fitness is the sum of the fitness of all widgets in p_n, that is $\sum_{w_u^n \in p_n} Fit(w_u^n)$, if w_v^m in p_m is not the widget that can trigger the interface jump, its fitness is N.

C. Generate Test Script

When the calculation of fitness for widgets is completed, automated testing can be guided with the fitness to reproduce the crashes more quickly. During testing, our method continuously selects and operates widgets in current interface of the application based on the fitness of widgets within a specified time limit. Specifically, we use formula (3) to calculate the fitness of any widget w_y^t within the current interface. In formula (3), $Fit(w_y^t)$ is the fitness of the widget w_y^t in current interface p_t, and $\sum_{w_z^t \in p_t} Fit(w_z^t)$ is the sum of the fitness of all widgets in p_t. For the widget $w_y^t \in p_t$, the probability of being selected to be operated is the ratio of its fitness to the sum of the fitness of all widgets in p_t. After selecting and operating a widget in current interface according to the probability, the operation is recorded in the test script. Our method ends the testing when the maximum test time is reached or the application crash is successfully reproduced, and outputs the test script. To determine whether the application crash has been successfully reproduced, the method compares the log with the stack trace in the bug report.

$$Pb(w_y^t) = \frac{Fit(w_y^t)}{\sum_{w_z^t \in p_t} Fit(w_z^t)} \quad (3)$$

III. Experiments and Evaluations

A. Experimental Design

Our method 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? We compare our method with PUMA and APE for crash reproduction. APE and PUMA are both model-based Android application testing tools. We measures effectiveness by the number and time costs to reproduce the crash described in the bug report within a limited time. The other settings of the experiment are as follows:

1) Experimental Settings: In the experiments, we follow the experimental settings of Zhao et al. [6] and limit the testing time of each experiment to 2 hours. To get more accurate time of reproducing crashes, each testing tool is set to run 10 times on each experimental subject. The average of the results will be recorded as the final time. In addition, after conducting small-scale experiments on our method, we set its parameter K to 45.

2) Experimental Subjects: The experimental subjects are obtained from Q-testing [7]. We analyze the above 50 applications in Q-testing. First, search for "Crash" on the bug-tracking systems of the 50 applications, and collect 27 bug reports. Then, manually verify and reproduce the crashes described in the bug reports, and remove the bug reports due to failure to build APK, environmental problems and unreproducible. As shown in TABLE I, six bug reports are obtained from five applications.

3) Implementation and Environment: Based on the proposed method, a prototype tool is implemented on the basis of PUMA framework to evaluate the effectiveness. The development and executing environment of the tool is 16GB memory, 6-core 3.3GHz CPU, Ubuntu20.04, Android SDK (4.3-8.0), JDK1.8.0.

B. Experimental Results

Fig. 2 shows the result of reproducing crashes by our method and automated testing tools PUMA, APE. Our method and APE can reproduce all crashes, but APE takes longer than our method. PUMA can reproduce five crashes, except for the crash in BetterBatteryStats. For these five crashes, our method and PUMA take same time to reproduce the crash in RadioBeacon (Bug1), while, our method takes less time to reproduce the other four crashes. Our method saves 71.47% time to reproduce crashes than PUMA in average. The reason is that the model-based exploration strategy of PUMA, which doesn't consider suspicious widgets during exploration, is less efficient in reproducing crashes. With further analysis of the crash in BetterBatteryStats which cannot be reproduces by PUMA, we found that PUMA triggered an exception during backtracking the application, which cause PUMA terminated and fail to reproduce the crash.

As shown in the experimental result, our method successfully reproduces the crashes in the bug reports, and outperforms PUMA and APE in terms of time costs.

C. Validity Analysis

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 our method 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 [7] [8], and the source codes are all open source on GitHub or F-Droid.

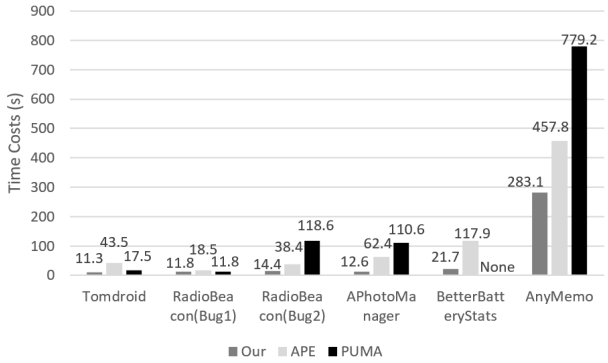

Fig. 2: The Comparison between our prototype tool, APE and PUMA on Crash Reproduction.

TABLE I: Basic Information of the Experimental Subjects.

Experimental Subjects	Link of 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

Bug reports are obtained from the respective bug tracking system or comment section of the application. To improve the generality of our method, we implement the method based on PUMA. PUMA has been validated on 3600 apps in Google Play [5] and is used as the base framework in the related works of Liu et al. [9].

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. FlowDroid is widely used for data flow analysis of Android applications and Java programs. PUMA framework is used to ensure the exploration of applications under test is accurate for our method. PUMA has been widely used in analyzing program attributes (such as application state, widget information, etc.) [5]. Moreover, we check and test the implementation code for constructing widget hierarchy graphs and calculating fitness of widgets to minimize the risk of validity.

IV. Related work

Currently, many works focused on GUI testing and the importance of bug reports in the quality assurance of Android applications.

Monkey [10] is the most commonly used random strategy based automated testing tool, which generates a pseudo-random stream of GUI events by randomly interacting with screen coordinates. The random strategy used by Monkey performs well on some benchmark applications. FUSION [11] assists users in automatically generating operation steps for reproducing bug reports by dynamically analyzing GUI events of Android applications. With FUSION, users can create more comprehensive and accurate bug reports, and developers can get operable information from bug reports, which could help reproduce and fix Android application bugs.

V. Conclusion

In this paper, we present a widget hierarchy graph guided crash reproduction method for Android apps. It creates a widget hierarchy graph by analyzing the project file and uses it with bug reports to generate test scripts for reproducing crashes. Experimental results show our method outperforms PUMA and APE in time costs. In the future, we plan to use information retrieval-based bug localization to improve the efficiency of reproducing crashes.

Aknowledgement

This work was supported in part by the Jiangsu Provincial Frontier Leading Technology Fundamental Research Project (BK20202001), the National Natural Science Foundation of China (No. 61702041), and the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project (No. QXTCP C201906).

References

[1] StatCounter, "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide/2022, 2023.

[2] APPLAUSE, "88% of people will abandon an app because of bugs," https://www.applause.com/blog/app-abandonment-bug-testing, 2017.

[3] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE international conference on software testing, verification and validation (icst)*. IEEE, 2016, pp. 33–44.

[4] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[5] 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 *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 204–217.

[6] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 128–139.

[7] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[8] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[9] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, 2015, pp. 89–103.

[10] Android Developers, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, 2021.

[11] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 673–686.