SCIENCE CHINA
Information Sciences

• HIGHLIGHT •

July 2017, Vol. 60 078102:1–078102:3
doi: 10.1007/s11432-015-0825-3

# Driving Android apps to trigger target API invocations based on activity and GUI filtering

Hongzhou YUE[1], Yuqing ZHANG[1,2]*, Wenjie WANG[2,3] & Qixu LIU[2,3]

[1]*State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an* 710071*, China;*
[2]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing* 101408*, China;*
[3]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093*, China*

Received June 1, 2016; accepted August 12, 2016; published online November 9, 2016

**Citation** Yue H Z, Zhang Y Q, Wang W J, et al. Driving Android apps to trigger target API invocations based on activity and GUI filtering. Sci China Inf Sci, 2017, 60(7): 078102, doi: 10.1007/s11432-015-0825-3

For an Android app tester, it is often needed to trigger the specific behaviors of an app. However, most of the existing Android GUI exploration tools only pursue high code coverage, rather than triggering the specific behaviors of an app [1, 2]. It inevitably leads to the unnecessary time consumption in the exploration process of an app if the testers only care about some specific behaviors of this app. Actually, methods of making a more targeted GUI exploration tool had been studied by some researchers [3–5], but for the reasons of time-consuming, low performance, high error rate, etc., none of these methods is practical enough for the testing work of large amounts of apps. Therefore, we proposed a more practical and effective GUI exploration tool—OA3E, which can drive a running app to trigger the specific behaviors which the testers are most interested in. We consider that API invocation is the intrinsic expression of the program behavior. So our tool takes the specific API invocations as input and drives a running app under test to trigger all these API invocations in it.

OA3E is based on A3E [6], an open-source and state-of-the-art Android GUI exploration tool. The main principle of A3E is systematically exploring an app to exercise any activity and method it can reach, trying best to improve code coverage. To make A3E be more focused on triggering the target API invocations, OA3E makes some major improvements on A3E. Two static analysis methods—activity filtering and GUI filtering—are applied to judge whether an activity or GUI element needs to be exercised. During the dynamic exploration process, OA3E only exercises the activities and GUI elements of an app that can lead to target API invocations, and avoids wasting time on those that cannot lead to target API invocations.

*Activity filtering.* For an Android app to be tested, $A^+$ is defined as the set of activities that need to be exercised in this app and $A^-$ is the opposite. Set $A$ is defined as all the activities in this app. Obviously, we have $A = A^+ + A^-$. Extending activity to the three types of Android components—activity, service, and broadcast receiver, we have $C^+$, $C^-$, $C$ and the relationship $C = C^+ + C^-$. For an activity $a \in A^+$, $G_a^+$ is defined as the set of GUI elements that need to be exercised and $G_a^-$ is the opposite. Set $G_a$ is defined as all the GUI elements in activity $a$.

Under the circumstance of not starting a new

* Corresponding author (email: zhangyq@ucas.ac.cn)
The authors declare that they have no conflict of interest.

component, if exercising component $c$ can lead to target API invocation $i$, we call component $c$ can reach $i$ inside the component, represented by $c \gg i$. Similarly, $c \gg j$ represents that $c$ can reach a component start invocation $j$ (e.g., "startActivity") inside the component. As an extension, $c \gg I$ and $c \gg J$ represent that $c$ can lead to a set of target API invocations and a set of component start invocations respectively. For each component start invocation $j$, $c_j$ is defined as the target component started by $j$ and $c \xrightarrow{j} c_j$ or $c \to c_j$ represents this component start relationship.

For each component in an Android app to be tested, whether it needs to be exercised can be determined by Theorem 1.

**Theorem 1.** For a component $c \in C$, known $c \gg I$, $c \gg J$, the following three rules are used to determine whether $c$ is in $C^+$ or $C^-$:

(1) If $I \neq \emptyset$, then $c \in C^+$.

(2) If $I = \emptyset \wedge J = \emptyset$, then $c \in C^-$.

(3) Known $I = \emptyset \wedge J \neq \emptyset$, if $\forall j \in J$, $c_j \in C^-$, then $c \in C^-$. Otherwise, $c \in C^+$.

After $C^+$ and $C^-$ are obtained, $A^+$ and $A^-$ which are the results of Activity Filtering can be got by $A^+ = \{c \mid c \in C^+ \text{ and } c \text{ is an activity}\}$ and $A^- = \{c \mid c \in C^- \text{ and } c \text{ is an activity}\}$.

*GUI filtering.* We use $g$ to represent a GUI element and $h$ to represent a GUI event handler. $h = H(g)$ is used to represent that $h$ is the event handler of $g$. Under the circumstance of not starting a new component, if exercising event handler $h$ can lead to target API invocation $i$, we call event handler $h$ can reach $i$ inside the component, represented by $h \gg i$. Similarly, $h \gg j$ represents that $h$ can reach a component start invocation $j$ inside the component. As an extension, $h \gg I$ and $h \gg J$ represent that $h$ can lead to a set of target API invocations and a set of component start invocations respectively. For each component start invocation $j$, $h \xrightarrow{j} c_j$ or $h \to c_j$ represents that component $c_j$ can be started by $h$.

For an activity $a \in A^+$, $H_a^+$ is defined as the set of GUI event handlers that need to be exercised in activity $a$ and $H_a^-$ as the set of GUI event handlers that need not. Set $H_a$ is defined as all the GUI event handlers in activity $a$. Obviously, we have $H_a = H_a^+ + H_a^-$. The construction method of $H_a^+$ and $H_a^-$ is shown by Theorem 2.

**Theorem 2.** For a GUI event handler $h \in H_a$, known $h \gg I$, $h \gg J$, the following three rules are used to determine whether $h$ is in $H_a^+$:

(1) If $I \neq \emptyset$, then $h \in H_a^+$.

(2) If $I = \emptyset \wedge J \neq \emptyset$, and $\exists j \in J$, make $c_j \in C^+$, then $h \in H_a^+$.

(3) Known $h' \in H_a^+$, if there exists an object be used in the path of $h' \gg I$ or $h' \gg J$, and its value can be changed by $h$, then $h \in H_a^+$.

After each GUI event handler in $H_a$ is processed according to Theorem 2, OA3E can get $H_a^+$. Then $H_a^-$ can also be got by $H_a^- = H_a - H_a^+$. With $H_a^+$ and $H_a^-$, $G_a^+$ and $G_a^-$ can be constructed by Theorem 3.

**Theorem 3.** For a GUI element $g \in G_a$, the following two rules are used to determine whether $g$ is in $G_a^+$ or $G_a^-$:

(1) If $\exists h = H(g)$, and if $h \in H_a^+$, then $g \in G_a^+$, else if $h \in H_a^-$, then $g \in G_a^-$.

(2) If $\exists h \in H_a^+$, and the value of GUI object corresponding to $g$ is used by $h$, then $g \in G_a^+$.

For a GUI element $g$ in activity $a$, we define $f$ as the GUI feature combination of $g$, represented by $f = F(g)$. If $f$ can uniquely identify $g$ in activity $a$, we call $g$ is unique. $F_a^+$ and $F_a^-$ are defined as the sets of GUI unique features combinations of GUI elements in activity $a$ that need to be exercised and need not respectively, namely, $F_a^+ = \{f \mid f = F(g),\ g \in G_a^+ \text{ and } f \text{ is unique}\}$ and $F_a^- = \{f \mid f = F(g),\ g \in G_a^- \text{ and } f \text{ is unique}\}$.

OA3E mainly cares about three types of features—view ID, displayed text and GUI element type. We define these features as $\alpha$, $\beta$ and $\gamma$ respectively. A GUI element can be uniquely identified by the following three feature combinations: $\langle\alpha,\gamma\rangle$, $\langle\beta,\gamma\rangle$ and $\langle\gamma\rangle$.

Not all GUI elements can be found their corresponding unique features. To solve this problem, OA3E creates a boolean variable "exerciseOther" for each activity. If the value of "exerciseOther" for an activity $a$ is "true", it means that except for the GUI elements whose feature combinations are in $F_a^+$, some other GUI elements in this activity also need to be exercised, but OA3E cannot identify them. On this condition, OA3E needs to exercise any GUI elements in this activity except for the GUI elements that possess features in $F_a^-$. On the contrary, if the value of "exerciseOther" is "false", it means that all the GUI elements in $G_a^+$ can be found the corresponding feature combinations in $F_a^+$, OA3E only needs to exercise the GUI elements that possess features in $F_a^+$ for activity $a$.

After the process of GUI Filtering, OA3E builds a map $\text{Mapa} = \{\langle a, (F_a^+, F_a^-, \text{exerciseOther})\rangle \mid a \in A^+\}$, "$a$" means one of the activities in $A^+$ and the three tuple $(F_a^+, F_a^-, \text{exerciseOther})$ is its corresponding GUI elements information which can tell OA3E's Automatic Explorer that in this activity, which GUI elements need to be exercised and which need not.

**Algorithm 1** OA3E's targeted and oriented exploration

```
Require: SATG $S = (A, E), A^+$, Mapa;
 1: Procedure TAOE($S$)
 2: for each node $a_i$ in $A$ that is entry activity and $a_i$ is in $A^+$ do
 3:     switch to activity $a_i$;
 4:     current$A \leftarrow a_i$;
 5:     for each edge $a_i \rightarrow a_j$ in $E$ do
 6:         if $a_j$ is an exported activity and $a_j$ is in $A^+$ then
 7:             switch to activity $a_j$;
 8:             current$A \leftarrow a_j$;
 9:             $S' \leftarrow$ get subgraph of $S$ from $a_j$;
10:             TAOE($S'$);
11:         end if
12:     end for
13:     $G_a \leftarrow$ extract GUI elements of current$A$;
14:     Get ($F_a^+, F_a^-$, exerciseOther) from $Mapa$;
15:     for each $g_i$ in $G_a$ do
16:         if (exerciseOther is false and $F(g_i) \in F_a^+$) or (exerciseOther is true and $F(g_i) \notin F_a^-$) then
17:             exercise $g_i$;
18:             if the current activity jumps to a not-yet-explored activity $a_k$ and $a_k$ is in $A^+$ then
19:                 $S' \leftarrow$ get subgraph of $S$ from $a_k$;
20:                 current$A \leftarrow a_k$;
21:                 TAOE($S'$);
22:             end if
23:         end if
24:     end for
25: end for
```

*OA3E and GUI exploration.* After the process of Activity Filtering and GUI Filtering, the following information can be obtained: $A^+$, $A^-$ and the map Mapa. It can be considered that activities in $A^+$ need to be exercised and activities not in $A^+$ need not to be exercised. For each activity $a \in A^+$, OA3E takes its corresponding three tuple $(F_a^+, F_a^-, \text{exerciseOther})$ from Mapa. If "exerciseOther=false", OA3E considers that GUI elements that possess features in $F_a^+$ need to be exercised. Else if "exerciseOther=true", it considers that GUI elements that do not possess features in $F_a^-$ need to be exercised. OA3E's exploration algorithm is shown in Algorithm 1. In this algorithm, $A^+$ and Mapa are defined as static objects which can be used in the whole process of the algorithm. In lines 2, 6 and 18, the same judgment is added to judge whether the target activity is in $A^+$. If the target activity is in $A^+$, it needs to be exercised. Otherwise, it needs not to be exercised. In this way, OA3E can avoid wasting time on exercising the activities that cannot lead to the target API invocations.

*Evaluation.* 100 popular apps from Google Play were randomly downloaded to test OA3E's performance. We had mainly done two experiments to evaluate OA3E's effectiveness and efficiency in triggering the target API invocations. In the first experiment, we used 21 APIs which require permission "INTERNET" or "READ_PHONE_STATE" to be invoked as the target APIs to test these 100 apps. OA3E and A3E were used to test each of the 100 apps once respectively, and a results comparison between the two tools had been made. Results showed that OA3E is more efficient in triggering the target API invocations than A3E. The average test time of OA3E is only 44.79% of A3E's. In the second experiment, one of the 100 apps was choosed to study the factors that affect OA3E's performance. Results showed that OA3E's total test time is positively related to the target APIs number and size of $A^+$ of each app. Therefore, if OA3E considers less APIs, or the distribution of these APIs in the app under test is more concentrated, then OA3E would cost less test time and be more efficient than A3E.

**Acknowledgements** This work was supported by National Natural Science Foundation of China (Grant Nos. 61572460, 61272481, 61303239) and Open Project Program of the State Key Laboratory of Information Security (Grant Nos. 2015-MS-06, 2015-MS-04).

**Supporting information** The supporting information is available online at info.scichina.com and link.springer.com. The supporting materials are published as submitted, without typesetting or editing. The responsibility for scientific accuracy and content remains entirely with the authors.

## References

1 Amalfitano D, Fasolino A R, Tramontana P, et al. Using GUI ripping for automated testing of Android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, 2012. 258–261

2 Machiry A, Tahiliani R, Naik M. Dynodroid: an input generation system for android apps. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, 2013. 224–234

3 Bhoraskar R, Han S, Jeon J, et al. Brahmastra: Driving apps to test the security of third-party components. In: Proceedings of the 23rd USENIX Conference on Security, San Diego, 2014. 1021–1036

4 Zheng C, Zhu S, Dai S, et al. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Raleigh, 2012. 93–104

5 Wong M Y, Lie D. IntelliDroid: a targeted input generator for the dynamic analysis of Android malware. In: Proceedings of the 23rd Network and Distributed System Security Symposium, San Diego, 2016

6 Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of android apps. ACM SIGPLAN Notices, 2013, 48: 641–660