SCIENCE CHINA
Information Sciences

• **Supplementary File** •

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*

Appendix A Introduction of A3E and Targeted Exploration

A3E is an automatic Android GUI exploration tool which uses two GUI exploration strategies—targeted exploration and depth-first exploration [1]. Depth-first exploration regards the entry point activities of an app as the starting points of the exploration and the exploration is guided by a depth-first traversal algorithm. Targeted exploration makes full use of the exported activities and can start the exported activities directly. Through the experiments, the paper [1] verified that targeted exploration is more efficient than depth-first exploration. So we mainly concern about the targeted exploration of A3E.

Figure B1 shows the workflow of A3E's targeted exploration. The input of A3E is app's bytecode. A3E uses SCanDroid [2] to construct SATG (Static Activity Transition Graph) which reflects the transition relationships between the activities. Then A3E uses the Automatic Explorer to explore the app through the guidance of SATG.

Algorithm A1 A3E's targeted exploration

Input: SATG $S = (A, E)$;
Procedure TExploration(S)
for each node a_i in A that is entry activity **do**
  switch to activity a_i;
  $currentA \leftarrow a_i$;
  for each edge $a_i \rightarrow a_j$ in E **do**
    if a_j is an exported activity **then**
      switch to activity a_j;
      $currentA \leftarrow a_j$;
      $S' \leftarrow$ get subgraph of S from a_j;
      TExploration(S');
    end if
  end for
  $G_a \leftarrow$ extract GUI elements of $currentA$;
  for each g_i in G_a **do**
    exercise g_i;
    if the current activity jumps to a not-yet-explored activity a_k **then**
      $S' \leftarrow$ get subgraph of S from a_k;
      $currentA \leftarrow a_k$;
      TExploration(S');
    end if
  end for
end for

* Corresponding author (email: zhangyq@ucas.ac.cn)

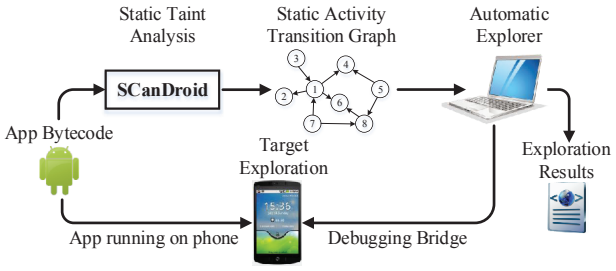

Figure B1 Overview of targeted exploration in A3E

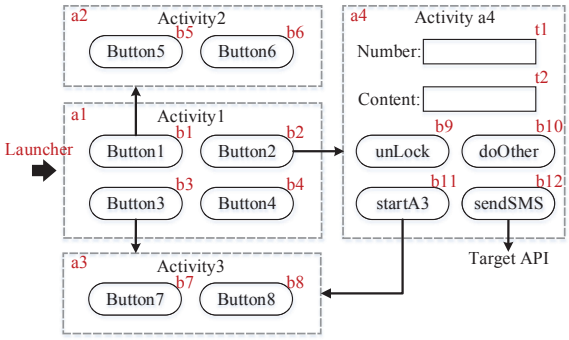

Figure B2 GUI architecture of an app. The red color texts are used to identify each element, which are not part of the app.

Algorithm A1 is the exploration algorithm of A3E's targeted exploration. The input of the algorithm is SATG $S = (A, E)$. For each node a_i in A that is entry activity, A3E finds each edge $a_i \rightarrow a_j$ in E. If a_j is an exported activity, it means that this activity can be independently called by A3E. Then A3E gets the subgraph S' of S from the starting node a_j. S' is used as input of "$TExploration$" which will be started as a new procedure. Subsequently, Automatic Explorer will extract and systematically exercise all the GUI elements in the current screen. If the current activity switches to a not-yet-explored activity a_k, A3E gets the subgraph S' of S from the starting node a_k. Then S' will be used as input of "$TExploration$" which will be started as a new procedure.

Appendix B Problem and Solution

Appendix B.1 Problem of A3E

Now let us go back to see what we need—we need a GUI exploration tool that can automatically drive a running app to trigger the target API invocations in it. Obviously A3E can help us accomplish this task, because A3E can systematically explore an app to exercise any activity and method it can reach. But the problem is—if we just want A3E to trigger some API invocations that we are interested in, is A3E efficient?

We can illustrate this problem by a simple example shown in Figure B2. Figure B2 shows the structure of an app which has four activities—$a1$, $a2$, $a3$ and $a4$. Suppose that we only care about the target API invocation "$sendTextMessage$", which only exists in activity $a4$. Activity $a4$ shows a screen to send text message. After input number in editText $t1$ and content in editText $t2$ and click button $b9$ to unlock the function of sending message, a click of button $b12$ will trigger the target API invocation "$sendTextMessage$" and send out a text message. Activity $a1$ is the launcher activity of this app, which is composed of four buttons. Click of buttons $b1$, $b2$, $b3$ of $a1$ will start activities $a2$, $a4$, $a3$ respectively.

If we use A3E to explore this app, it will exercise every activity and every GUI element in each activity. Namely it will exercise activities $a1 \sim a4$ and GUI elements $b1 \sim b12$, t1, t2. However, because we only care about the target API invocations. So we only need A3E to exercise activities $a1$, $a4$ and GUI elements $b2$, $b9$, $b12$, $t1$, $t2$, not the others. From this point of view, we think that A3E's exploration strategy is inefficient.

Appendix B.2 Our Solution

For convenience of description, we give the following definitions:

Definition 1. For an Android app to be tested, symbol A^+ is defined as the set of activities that need to be exercised in this app and A^- as the set of activities that need not. Set A is defined as all the activities in this app. Obviously, we have $A = A^+ + A^-$.

Definition 2. For an activity $a \in A^+$, symbol G_a^+ is defined as the set of GUI elements that need to be exercised and G_a^- as the set of GUI elements that need not. Set G_a is defined as all the GUI elements in activity a. Obviously, we have $G_a = G_a^+ + G_a^-$.

According to Definition 1 and 2, for the example shown in Figure B2, we can get $A^+ = \{a1, a4\}$, $A^- = \{a2, a3\}$, $G_{a1}^+ = \{b2\}$, $G_{a1}^- = \{b1, b3, b4\}$, $G_{a4}^+ = \{t1, t2, b9, b12\}$, $G_{a4}^- = \{b10, b11\}$.

As a matter of fact, to improve A3E's efficiency in triggering target API invocations, measures should be taken to construct A^+ and A^- of an app. Similarly, for each activity $a \in A^+$, measures should be taken to construct G_a^+ and G_a^- of activity a. In the exploration period of A3E, A3E should only exercise activities in A^+ and restrain from exercising those in A^-. Similarly, for each activity $a \in A^+$, A3E should only exercise activities in G_a^+ and restrain from exercising those in G_a^-.

Therefore, two static analysis methods—Activity Filtering and GUI Filtering are proposed in this paper to improve A3E's efficiency in triggering target API invocations. Activity Filtering is used to construct A^+ and A^- of an app, while GUI Filtering is used to construct G_a^+ and G_a^- for each $a \in A^+$. We made several improvements on A3E and integrated these two static analysis methods into the static analysis module of A3E, and proposed a new Android GUI exploration

tool—OA3E (Oriented A3E). Compared with A3E, OA3E's advantage is that it can focus on exercising the activities and GUI elements that need to be exercised and avoid wasting time on exercising the items that cannot lead to the target API invocations. It can greatly improve the efficiency of A3E's exploration process, which will be verified in Section Appendix F. The implementation details of Activity Filtering and GUI Filtering will be introduced in Section Appendix C and Appendix D.

Appendix C Activity Filtering

Now we will discuss how OA3E implements Activity Filtering to construct the activities set A^+ and A^- of an app.

Appendix C.1 Main Idea

For convenience of description, we assume that the code fragments shown in Listing 1 and 2 are the Java code and the corresponding layout XML code of Activity a4 in Figure B2 respectively. Besides, we give Definition 3 and 4:

Listing 1 Code fragment of Activity a4

```
EditText number, content;
Button unLock, doOther, send;
boolean  lock;
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    lock = true;
    number = (EditText) findViewById(R.id.number); // t1
    content = (EditText) findViewById(R.id.content); // t2
    unLock = (Button) findViewById(R.id.unLock); // b9
    doOther = (Button) findViewById(R.id.doOther); // b10
    send = (Button) findViewById(R.id.send); // b12
    unLock.setOnClickListener(new OnClickListener() {
        public void onClick(View arg0) { // h1
            lock = false;
    }});
    doOther.setOnClickListener(new OnClickListener() {
        public void onClick(View arg0) { // h2
            /* do Other thing */
    }});
    send.setOnClickListener(new OnClickListener() {
        public void onClick(View arg0) { // h3
            if (!lock) {
                PendingIntent pi = PendingIntent.getActivity(ActivityA4.this, 0,
                    new Intent(), 0);
                SmsManager.getDefault().sendTextMessage(number.getText().toString()
                    , null, content.getText().toString(), pi, null); // i1
    }}});
}
public void clickHandler(View source) { //h4, binded handler of b11
    ComponentName comp = new ComponentName(ActivityA4.this, ActivityA3.class);
    Intent intent = new Intent();
    intent.setComponent(comp);
    startActivity(intent); // j1
}
```

Listing 2 Layout XML fragment of Activity a4. The ellipses signify code omitted for the sake of clarity.

```
<EditText android:id="@+id/number" .../>
<EditText android:id="@+id/content" .../>
<Button android:id="@+id/unLock" android:text="unLock" .../>
<Button android:id="@+id/doOther" android:text="doOther" .../>
<Button android:onClick="clickHandler" android:text="startA3" .../>
<Button android:id="@+id/send" android:text="sendSMS" .../>
```

Definition 3. For an Android app to be tested, symbol C^+ is defined as the set of components that need to be exercised in this app and C^- as the set of components that need not. Set C is defined as all the components in this app. Obviously, we have $C = C^+ + C^-$.

The component mentioned in Definition 3 refers to one of the three components of Android—activity, service, and broadcast receiver [3]. They can be started by some specific API invocations. For instance, an activity can be started by invoking method "*startActivity*" or "*startActivityForResult*". We call this type of invocation "component start invocation". As activity is one of the three components, so we can get $A \subseteq C$. The reason why we have to consider the three types of

components is that components can start each other. For instance, an activity may start any other components, in which may also exist target API invocations.

Definition 4. We use symbol i to represent a target API invocation and I to represent a set of target API invocations. Symbol j is used to represent a component start invocation and J represents a set of component start invocations. Under the circumstance of not starting a new 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, under the circumstance of not starting a new component, if exercising component c can lead to component start invocation j, we have $c \gg j$. 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, symbol c_j is defined as the target component started by j and $c \xrightarrow{j} c_j$ or $c \rightarrow c_j$ represents this component start relationship.

Take the code fragment shown in Listing 1 as an example, there is a target API invocation named "$sendTextMessage$" in line 25. We use $i1$ to represent this target API invocation for convenience of description. A component start invocation named "$startActivity$" can also be found in line 32. We use $j1$ to represent this component start invocation. In combination with Figure B2, we can get $a4 \gg i1$, $a4 \gg j1$ and $c_{j1} = a3$.

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, $c \in C^+$.

Theorem 1.(1) means that, if component c can reach some target API invocations inside the component, then $c \in C^+$. There is no doubt about it. For example, Component (Activity) $a4$ shown in Figure B2 is a component of this type and it can reach the target API invocation $i1$ inside the component. Namely, $a4 \in C^+$. Theorem 1.(2) means that, if component c can neither reach any target API invocation nor any component start invocation inside the component, then $c \in C^-$. There is also no doubt about it because in this case, component c can neither reach any target API invocation inside the component, nor reach any target API invocation indirectly by starting and exercising a new component. For example, Component (Activity) $a2$ and $a3$ shown in Figure B2 are components of this type. Namely, $a2, a3 \in C^+$. Theorem 1.(3) means that, if component c cannot reach any target API invocation inside the component, but can reach some component start invocations J inside the component, whether c is in C^+ or C^- depend on the components started by J. If all the components started by J are in C^-, it means that component c cannot reach any target API invocation indirectly by starting and exercising a new component, then $c \in C^-$. Otherwise, if $\exists j \in J$, $c_j \in C^+$, it means that component c can reach some target API invocations indirectly by starting and exercising a new component. For example, Component (Activity) $a1$ shown in Figure B2 is a component of this type and it can reach the target API invocation $i1$ indirectly by starting and exercising activity $a4$. Namely, $a1 \in C^+$.

Appendix C.2 Implementation

OA3E judges whether a component can reach a target API invocation or component start invocation inside the component by static analysis of the control flow graph of the whole app built by WALA [4]. Directed by the control flow graph, if a component c can reach a target API invocation i or component start invocation j, it can be got $c \gg i$ or $c \gg j$. To address the challenge of implicit control flow transitions through the Android framework, OA3E incorporates the results generated by EdgeMiner [5] to achieve a more complete modeling of Android callback mechanism and a more complete control flow graph.

In Section Appendix A, we had introduced that A3E constructs SATG to guide its targeted exploration. SATG built by A3E has described the transition relationships between the activities. As an extension, OA3E takes into account the transition relationships between the three types of Android components, not just activities. Accordingly, OA3E construct a new graph—SCTG (Static Component Transition Graph), and the construction method of SCTG is the same as SATG. Both of them use static taint analysis over the intent passing logic to find the relationship between the target API invocation j and the target component c_j.

Based on SCTG, OA3E takes a graph analysis algorithm to determine whether a component c is in C^+ or C^-, this algorithm can be simply described as four steps: Step 1, a depth-first traversal algorithm [6] is used to change SCTG to a directed acyclic graph [7]. Step 2, initialize set C to be all of the components in SCTG, and $C^+ = \emptyset$, $C^- = \emptyset$. Step 3, pull the components that satisfy Theorem 1.(1) and 1.(2) out of C, and put them into C^+ and C^- respectively. Step 4, pull the components that satisfy Theorem 1.(3) out of C, and put them into C^+ or C^- according to the actual situation. Repeat this step until $C = \emptyset$.

Figure C1 is used to illustrate this analysis procedure of an SCTG. The red nodes represent components that need to be exercised, the blue nodes represent components that need not to be exercised, and the white nodes represent the components that need to be judged whether they need to be exercised. Component $c1$ represents the launcher activity or exported activity which can act as the starting point of targeted exploration of A3E or OA3E. $c6$ represents a component that satisfies Theorem 1.(1), while $c7$ and $c8$ represent components that satisfy Theorem 1.(2). Step 1 is shown by the conversion from Figure C1.(a) to C1.(b). The reason why OA3E implements Step 1 is that, in the process of dynamic exploration of an app, OA3E needs not to exercise an activity that had been exercised. Or it can be said that, if $c \xrightarrow{j} c_j$ and $c_j \in C^+$, but c_j is exercised before c according to A3E's targeted exploration strategy, then $c \in C^+$ cannot be got.

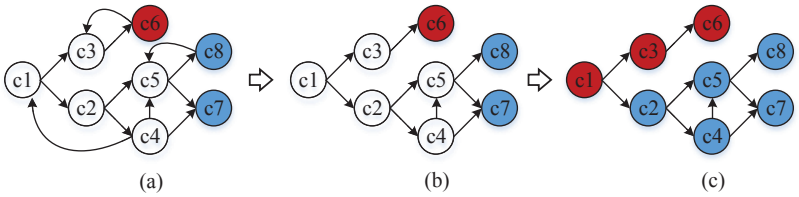

Figure C1 Graph analysis procedure of an SCTG

Table D1 The relationship between GUI element, event handler and concerned API invocation of Activity a4

GUI Element	GUI Object	Event Handler	Concerned API Invocation
b9	Button *unLock*	h1: onClick (Line 14)	—
b10	Button *doOther*	h2: onClick (Line 18)	—
b11	—	h4: clickHandler (Line 28)	j1: startActivity (Line 32)
b12	Button *send*	h3: onClick (Line 22)	i1: sendTextMessage (Line 25)

Therefore, this condition should be excluded by Step 1 to ensure the accuracy of analysis. For example, in Figure C1(a), $c4 \rightarrow c1$ and $c1 \in C^+$, but $c4 \in C^+$ cannot be got. In Step 3, $c6$ which satisfies Theorem 1.(1) was put into into C^+, while $c7$ and $c8$ which satisfy Theorem 1.(2) were put into into C^-. In Step 4, $c5, c4, c2$ were put into C^- in turn, while $c1, c3$ were put into C^+. At last, it can be got that, $C^+ = \{c1, c3, c6\}$ and $C^- = \{c2, c4, c5, c7, c8\}$.

There is an important problem in the process of constructing SCTG. That is, the completeness of SCTG sometimes cannot be guaranteed. We had pointed out that the construction method of SCTG is the same as SATG which had been realized by A3E. But A3E itself cannot guarantee the completeness of SATG, because sometimes the taint analysis method that A3E uses to construct SATG cannot guarantee that the target component corresponding to a component start invocation can be found. This drawback also exists in the construction of SCTG, namely, there exists the case that, known $c \gg j$, but c_j cannot be got. In this case, to reduce error, OA3E's strategy is that if there is a component start invocation j in an component c that cannot be found the target component c_j, OA3E considers that $c \in C^+$.

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

Appendix D GUI Filtering

Now we will discuss how OA3E implements GUI Filtering to find G_a^+ and G_a^- for each $a \in A^+$. OA3E mainly does five steps to implement GUI Filtering which will be introduced in the following sections from Appendix D.1 to Appendix D.5.

Appendix D.1 Search GUI Elements and Their Corresponding Event Handlers

Definition 5. We use symbol 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.

GUI event handler can be realized by two ways—event listener and method binding [8]. For GUI event handlers realized by method binding, OA3E can easily find the corresponding handlers from the layout XML file. For event handlers realized by event listener, OA3E uses static taint analysis method which is based on SCanDroid to search the GUI elements and their corresponding event handlers. OA3E marks each GUI object declaration as source and the event handler registration API as sink. The code shown in Figure D1 is used to illustrate this taint analysis process. In Figure D1, button "*a*" is marked as source and the event handler registration API "*setOnClickListener*" is marked as sink. If SCanDroid can find a path from the source to the sink, then OA3E considers that it had found the relationship between a GUI element and its corresponding event handler (Look at the blue dotted arrow in Figure D1).

OA3E also gets the GUI element type and event type from the layout XML file as well as from the code. The former method is easy to implement by reading the properties of an GUI element from the layout XML file. For the latter method, we illustrate it using the code shown in Figure D1. It can be easily seen that the type of GUI element "*a*" is button and the corresponding event type is "*Click*" which is known from the event handler registration statement.

Let's go back to see activity $a4$ shown in Figure B2 combined with Listing 1 and 2. After completing the search work for the GUI elements and their corresponding event handlers, OA3E can get the results shown in the first 3 columns of Table D1 for activity $a4$. The relationship between the GUI elements and their corresponding event handlers are: $h1 = H(b9)$, $h2 = H(b10)$, $h4 = H(b11)$ and $h3 = H(b12)$.

Appendix D.2 Search Relationships Between Event Handlers and Concerned API Invocations

Definition 6. 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, under the circumstance of not starting a new component, if exercising event handler h can lead to component start invocation j, we

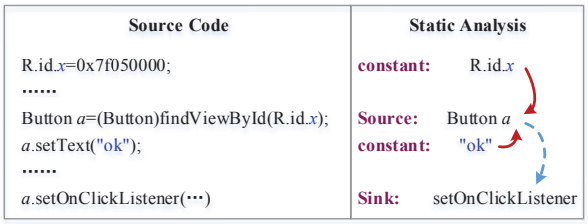

Figure D1 Static analysis process of GUI filtering

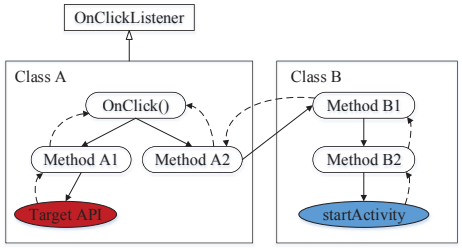

Figure D2 Concerned API invocation searching

have $h \gg j$. 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 $h \rightarrow c_j$ represents that component c_j can be started by h.

Searching backward along the control flow graph built by WALA, OA3E can build the relationships between GUI event handlers and the concerned API invocations (including target API invocations and component start invocations). This process can be described by the example shown in Figure D2. In Figure D2, there are two classes—class A and B. Class A is a GUI event listener class containing an event handler method "*OnClick*" inside. The event handler method calls method $A1$ that can lead to the target API invocation. It also calls method $A2$ of class A and method $B1, B2$ of class B which can lead to the component start invocation "*startActivity*". Started from these two concerned API invocations, OA3E searches backward along the control flow graph until reaching "*onClick*" method (The dotted arrows in Figure D2 represent the search trajectories). Then the relationships between event handler and the two concerned API invocations are constructed.

For activity $a4$ shown in Figure B2, combined with Listing 1 and 2, a target API invocation $i1$ and a component start invocation $j1$ can be found, and the relationships between them with the corresponding event handlers $h3$ and $h4$ are listed in Table D1. That is, it can be got $h3 \gg i1$ and $h4 \gg j1$.

Appendix D.3 Judge Whether a GUI Element Needs to be Exercised

In order to facilitate the description of the method of judging whether a GUI element needs to be exercised, we give Definition 7:

Definition 7. For an activity $a \in A^+$, symbol 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 $H_a = H_a^+ + H_a^-$.

For each activity $a \in A^+$, G_a and H_a had been obtained in Section Appendix D.1. Before OA3E constructs set G_a^+ and G_a^-, set H_a^+ and H_a^- must be constructed first. The construction method of H_a^+ and H_a^- is shown by Theorem 2. 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 $H_a^- = H_a - H_a^+$.

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 being used in the path of $h' \gg I$ or $h' \gg J$, and its value can be changed by h, then $h \in H_a^+$.

Take the app shown in Figure B2 as an example, combined with Listing 1 and 2, it can be got by Theorem 2.(1) that event handler $h3 \in H_a^+$ because of $h3 \gg i1$. According to Theorem 2.(3), event handler $h1 \in H_a^+$ because the value of object "*lock*" used in the path of $h3 \gg i1$ can be changed by $h1$. According to Theorem 2.(2), it cannot be got $h4 \in H_a^+$ because $h \gg j1$, but $c_{j1} \in C^-$. On the contrary, the event handler (of which the code is not displayed) corresponding to GUI element $b2$ in activity $a1$ satisfies Theorem 2.(2) and needs to be exercised. After all the GUI event handlers in H_{a4} are processed according to Theorem 2, it can be got $H_{a4}^+ = \{h1, h3\}$ and $H_{a4}^- = \{h2, h4\}$.

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 activity $a4$ shown in Figure B2, it can be got $b9, b12 \in G_{a4}^+$ and $b10, b11 \in G_{a4}^-$ by Theorem 3.(1). Because the GUI objects "*number*" and "*content*" which corresponding to GUI elements $t1$ and $t2$ respectively are used in event handler $h3$ (line 25 in Listing 1), therefore $t1, t2 \in G_{a4}^+$. To summarize, it can be got $G_{a4}^+ = \{b9, b12, t1, t2\}$ and $G_{a4}^- = \{b10, b11\}$ for activity $a4$. Both Theorem 2.(3) and Theorem 3.(2) are related to value of an object. OA3E tracks the value of an object by the method of point-to analysis [9] and taint analysis.

Table D2 Results of GUI feature search for Activity a4

GUI Element	GUI Object	GUI Feature Combination			Belong
		View Id	Displayed Text	Type	
b9	Button *unLock*	*R.id.unLock*	*unLock*	Button	F_a^+
b10	Button *doOther*	*R.id.doOther*	*doOther*	Button	F_a^-
b11	—	—	*startA3*	Button	F_a^-
b12	Button *send*	*R.id.send*	*sendSMS*	Button	F_a^+
t1	EditText *number*	*R.id.number*	—	EditText	F_a^+
t2	EditText *content*	*R.id.content*	—	EditText	F_a^+

Appendix D.4 Search Features of GUI Elements

G_a^+ and G_a^- built in Section Appendix D.3 represent the GUI elements that need to be exercised and those need not respectively. However, they cannot guide the exploration process of OA3E's Automatic Explorer because Automatic Explorer cannot identify the GUI elements in G_a^+ and G_a^- during the running process of an app. So the features that can uniquely identify a GUI element and can be recognized by Automatic Explorer should be found. For convenience of description, we give Definition 8:

Definition 8. For a GUI element g in activity a, we define symbol 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, $F_a^+ = \left\{ f \mid f = F(g),\ g \in G_a^+ \textit{ and } f \textit{ is unique} \right\}$ and $F_a^- = \left\{ f \mid f = F(g),\ g \in G_a^- \textit{ and } f \textit{ is unique} \right\}$.

OA3E mainly cares about three types of features—view ID, displayed text and GUI element type. We define these features as α, β and γ respectively. Feature γ of a GUI element had been obtained in Section Appendix D.1. As for the other two features, OA3E uses two methods to search—searching from the layout XML file and searching from the code. For the first method, it is easy to read the features of a GUI element from the layout XML file, such as the attributes "$android:id$" and "$android:text$" shown in Listing 2. As for the second method, OA3E uses the method of tracing the propagation of constant string to find the view ID and displayed text of a GUI element. When a constant string propagates to one of the GUI elements in G_a^+ or G_a^-, OA3E judges whether it is the view ID or displayed text of this GUI element by the API name of the touch point. This process can be described by Figure D1. In Figure D1, the red solid arrows represent the propagation paths of constant strings. OA3E tracks the propagation of the constant strings "*R.id.x*" and "*ok*" until they reach the GUI element—button "*a*". Then OA3E judges the features of this GUI element by the API name of the touch point, such as API "$findViewById$" and "$setText$" in Figure D1. It can be got that "*R.id.x*" is the view ID of button "*a*", and "*ok*" is its displayed text.

A GUI element can be uniquely identified by the following three feature combinations:

(1) $\langle \alpha, \gamma \rangle$. In this case, OA3E's static analysis method can find the view ID of a GUI element. There is no doubt that this feature combination can uniquely identify a GUI element because view ID is unique for each GUI element.

(2) $\langle \beta, \gamma \rangle$. This case means that OA3E cannot find the view ID of a GUI element. But it can find the displayed text of the GUI element such as the displayed text "ok" in the code of Figure D1. After the static analysis, if OA3E finds that no other GUI elements in the activity that have the same displayed text as this GUI element, then it can be considered that this feature combination can uniquely identify a GUI element. Otherwise, it is not the unique feature of a GUI element.

(3) $\langle \gamma \rangle$. This case means that OA3E cannot find the view ID and displayed text of a GUI element. But OA3E finds that in this activity, no other GUI elements belong to this GUI element type. Namely, only GUI element type can identify this GUI element. In this case, the GUI element type is the unique feature of a GUI element.

For activity *a4* shown in Figure B2, the results of GUI feature search are shown in Table D2. All of the GUI feature combinations are unique and can be used to identify the GUI elements in G_{a4}^+ and G_{a4}^- .

Appendix D.5 Results of GUI Filtering

F_a^+ and F_a^- built in Section Appendix D.4 are used by OA3E to judge that, in activity a, GUI elements possess what features need to be exercised and what need not. With the help of the GUI feature combinations provided by F_a^+ and F_a^-, OA3E's Automatic Explorer can identify the GUI elements in G_a^+ and G_a^- during the running process of an app. However, not all GUI elements can be found their corresponding unique features. The reason is that the only feature combination can be found of a GUI element is the same as another GUI element in an activity. Therefore, $\exists g \in G_a^+$, $f = F(g)$, but as f is not unique, then $f \notin F_a^+$. In this case, it is incomplete for OA3E to only exercise the GUI elements that possess features in F_a^+. To solve this problem, OA3E creates a boolean variable "$exerciseOther$" for each activity, and if this case happens, "$exerciseOther$" will be assigned to "$true$". Otherwise, "$exerciseOther$" is assigned to "$false$".

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.

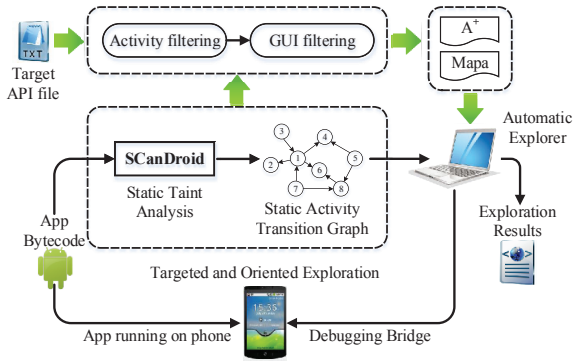

Figure E1 Overview of the exploration process of OA3E

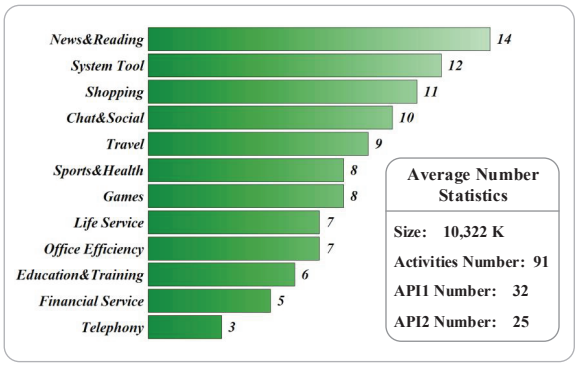

Figure E2 Basic information statistics of the 100 apps

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

Appendix E OA3E and GUI Exploration

In Section Appendix C and Appendix D, we had introduced that how OA3E implements Activity Filtering and GUI Filtering to solve the problems proposed in Section Appendix B. Now we will give the overview structure of OA3E and its GUI exploration algorithm.

Appendix E.1 Structure of OA3E

After the process of Activity Filtering and GUI Filtering, the following information can be obtained: A^+, A^- and the map $Mapa$. With this information, the problem proposed in Section Appendix B can be solved. 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 $\left(F_a^+, F_a^-, exerciseOther \right)$ 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.

At a result, we outline the overview structure of OA3E which is shown in Figure E1. It is based on A3E whose structure is shown in Figure B1 and has been made some improvements on it. The target APIs is defined in the "Target API file". It is the input of Activity Filtering and GUI Filtering of OA3E. The outputs A^+ and $Mapa$ will guide Automatic Explorer to explore an app which is running in the device or emulator. How Automatic Explorer is been guided will be introduced in the next section.

Appendix E.2 Targeted and Oriented Exploration

We modified A3E's targeted exploration algorithm described in Section Appendix A and proposed OA3E's exploration algorithm which is shown in Algorithm E1 and we call it "targeted and oriented exploration (TAOE)". In this algorithm, A^+ and $Mapa$ are defined as static objects which can be used in the whole process of the algorithm. In line 2, 6 and 18, the same judgment is been 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.

In line 14, OA3E gets the three tuple $\left(F_a^+, F_a^-, exerciseOther \right)$ corresponding to $currentA$ from $Mapa$. It can tell Automatic Explorer that which GUI elements in $currentA$ need to be exercised and which need not. The Troyd tool [10] which A3E's Automatic Explorer mainly depends on can be used to extract the GUI information of the current screen, including GUI coordinates, GUI type, view ID, displayed text, etc. OA3E compares this information with the GUI features in F_a^+ or F_a^- to identify the exact GUI element. The exploration strategy of OA3E is determined by the value of "$exerciseOther$". If "$exerciseOther = false$", OA3E will exercise the GUI elements that possess features in F_a^+. Else if "$exerciseOther = true$", OA3E will exercise the GUI elements that do not possess features in F_a^-.

Appendix F Evaluation

Appendix F.1 Experiment Method and Preparation

We define 5 sets of APIs in Table F1 which are of interest to us. Each set of APIs needs different permissions to be invoked. The last two columns list the class name and method name of each API. These APIs will be used as the inputs of OA3E (see the "Target API file" introduced in Section Appendix E.1). 100 apps were randomly selected and downloaded from

Algorithm E1 OA3E's targeted and oriented exploration

Input: SATG $S = (A, E), A^+, Mapa$;
Procedure TAOE(S)
for each node a_i in A that is entry activity and a_i is in A^+ **do**
  switch to activity a_i;
  $currentA \leftarrow a_i$;
  for each edge $a_i \rightarrow a_j$ in E **do**
    if a_j is an exported activity and a_j is in A^+ **then**
      switch to activity a_j;
      $currentA \leftarrow a_j$;
      $S' \leftarrow$ get subgraph of S from a_j;
      TAOE(S');
    end if
  end for
  $G_a \leftarrow$ extract GUI elements of $currentA$;
  Get $(F_a^+, F_a^-, exerciseOther)$ from $Mapa$;
  for each g_i in G_a **do**
    if ($exerciseOther$ is $false$ and $F(g_i) \in F_a^+$) or ($exerciseOther$ is $true$ and $F(g_i) \notin F_a^-$) **then**
      exercise g_i;
      if the current activity jumps to a not-yet-explored activity a_k and a_k is in A^+ **then**
        $S' \leftarrow$ get subgraph of S from a_k;
        $currentA \leftarrow a_k$;
        TAOE(S');
      end if
    end if
  end for
end for

Wandoujia [1)], a famous Android app store in China. These apps are distributed in 12 different categories and each of them is among the top 100 popular apps in their own categories according to the download ranking of Wandoujia. The left part of Figure E2 shows these categories and the number of apps in each category.

We mainly do two experiments to evaluate OA3E's effectiveness and efficiency in triggering the target API invocations. These two experiments will be introduced in Section Appendix F.2 and Appendix F.3 respectively. In the first experiment, we use APIs in set $API1$ and $API2$ of Table F1 as the target APIs to test these 100 apps. Set $API1$ contains 13 APIs and the invocations of these APIs require permission "INTERNET". Set $API2$ contains 8 APIs and the invocations of these APIs require permission "READ_PHONE_STATE". Our objective is to monitor app's behavior of reading the basic information of the phone and sending this information to the Internet. Sometimes these behaviors are related to privacy leakage of the phone users [11] [12]. A3E and OA3E will be used to test each of the 100 apps once respectively, and a results comparison between these two tools will be made. In the second experiment, one of the 100 apps will be used to test the efficiency of OA3E. Factors that affect OA3E's performance will be discussed, with the target APIs set being varied from $API1$ to API5 in the form of accumulation.

OA3E's main program runs in a desktop computer with Intel Core i7 CPU, 32G RAM and Windows 7 system. During the period of dynamic exploring, the app under test would be installed and run in a Samsung G9280 mobile phone with 8-core CPU, 4 GB RAM and Android OS 5.1.

Appendix F.2 Results Comparison Between A3E and OA3E

The first experiment was testing the 100 apps using APIs in set $API1$ and $API2$ as the target APIs. Before the experiment, we did an average number statistic for these 100 apps. The statistic results are shown in the right part of Figure E2. The average size of the 100 apps is 10,322K and the average activities number is 91. The static analysis method of searching along control flow graph (see the control flow graph introduced in Section Appendix C.2) was used to count the number of target API invocations in $API1$ and $API2$ of these 100 apps. The average number of target API invocations in $API1$ is 32, and for $API2$, the number is 25.

For each of the 100 apps, we use A3E and OA3E to run once respectively. The Xposed framework [13] was used to hook the targeted APIs in set $API1$ and $API2$. The API invocation information would be outputted to the log when the target API invocations happened. This information includes API names, callers' names (which can be obtained from the call stack), etc. Then the number of the triggered API invocations can be counted by the log.

Table F2 shows the overall test results for the 100 apps. A3E spent an average of 4 minutes on static analysis and 92 minutes on dynamic exploration for each app, then the average time A3E spent on each app is 96 minutes. For OA3E, the time costs are 13, 30 and 43 minutes for static analysis, dynamic exploration and the total time respectively. It can be noticed that the static analysis time of OA3E is more than A3E's. Obviously, it is because OA3E costs more time in the process of Activity Filtering and GUI Filtering, which does not exist in A3E. On the contrary, the dynamic exploration

1) https://www.wandoujia.com/

Table F1 The concerned target APIs

Set	Count	Permission	Class Name	API Name
*API*1	13	INTERNET	URL	openStream, openConnection, getContent
			URLConnection	getInputStream, connect
			DownloadManager	getUriForDownloadedFile, addCompletedDownload, enqueue
			ServerSocket	bind
			HttpURLConnection	connect
			WebViewFragment	onCreateView
			DefaultHttpClient	execute
			WebSettings	setBlockNetworkLoads
*API*2	8	READ_PHONE_STATE	TelephonyManager	getSubscriberId, getDeviceSoftwareVersion, listen, getLine1Number, getSimSerialNumber, getVoiceMailAlphaTag, getVoiceMailNumber, getDeviceId
*API*3	13	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION	LocationManager	requestLocationUpdates, getProviders, requestSingleUpdate, getProvider, getLastKnownLocation, isProviderEnabled, addProximityAlert, getBestProvider, sendExtraCommand, addNmeaListener, addGpsStatusListener
			TelephonyManager	getNeighboringCellInfo, getCellLocation
*API*4	7	ACCESS_NETWORK_STATE	ConnectivityManager	getNetworkInfo, getNetworkPreference, setNetworkPreference, stopUsingNetworkFeature, startUsingNetworkFeature, getActiveNetworkInfo, getAllNetworkInfo
*API*5	8	ACCESS_WIFI_STATE	WifiManager	getScanResults, getDhcpInfo, pingSupplicant, isWifiEnabled, getConnectionInfo, getWifiState, getConfiguredNetworks
			WifiP2pManager	initialize

Table F2 Statistics of test results for 100 apps

Statistical Content	Result
Average time of A3E (minutes)	Static time: 4 Dynamic time: 92 Total time: 96
Average time of OA3E (minutes)	Static time: 13 Dynamic time: 30 Total time: 43
Average Size of A^+	38
OA3E's accuracy of triggering target APIs compared to A3E	98.63%

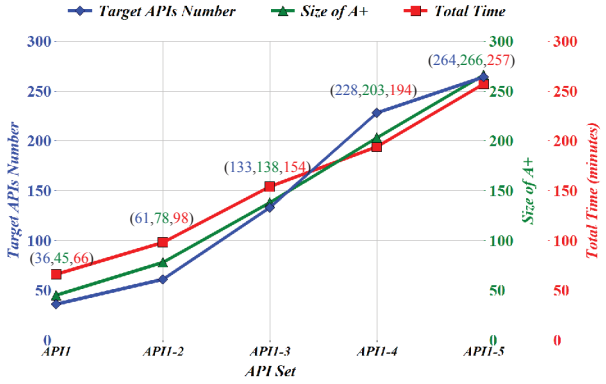

Figure F1 The relationship between target APIs number, Size of A^+ and total test time for app ρ10

time of OA3E is much less than A3E's. It is because OA3E does not need to exercise the activities and GUIs that cannot lead to the target API invocations. It saves a lot of time for OA3E in its dynamic exploration of the apps. It can also be clearly seen that the static analysis costs much less time than the dynamic exploration for each of the apps. Therefore, even if OA3E costs more time in static analysis, the total test time of OA3E for each app is still less than A3E. The average test time of OA3E is only 44.79% ($\frac{43}{96}$) of A3E's.

The average size of A^+ shown in Table F2 means that, on average, only 38 activities in each app need to be exercised. Compared with the average activities number shown in Figure E2, we found that OA3E's Activity Filtering function could filter out an average of 58.24% ($\frac{91-38}{91}$) activities that need not to be exercised with the target APIs in set *API*1 and *API*2 as inputs. Meanwhile, OA3E's accuracy of triggering target APIs is 98.63% compared to A3E, it means that 98.63% of the target APIs invocations that triggered by A3E can be successfully triggered by OA3E. According to our analysis, why OA3E cannot achieve 100% were mainly caused by the inaccurate static analysis in the process of Activity Filtering and GUI Filtering and the uncertain factors in dynamic exploration. These reasons will not be the focus of our discussion because it can be considered that 98.63% is already an acceptable ratio and some related discussions will be left in Section Appendix G.2. The reason why A3E and OA3E can trigger nearly the same proportion of the target API invocations is that, A3E and OA3E share the same GUI exploration algorithm (besides the GUI Filtering part), and they generate the

Table F3 Basic information of the 10 apps

APP Id	Package Name	Category	Size (K)	Activities Number	Target APIs Number ($API1 + API2$)
$\rho 1$	com.jiayuan	Chat&Social	6,388	112	34 (11+23)
$\rho 2$	com.fayi.law	Life Service	8,543	86	204 (132+72)
$\rho 3$	me.ele	Life Service	6,676	76	86 (40+46)
$\rho 4$	com.sohu.newsclient	News&Reading	13,725	96	139 (105+34)
$\rho 5$	com.tingwen	News&Reading	2,735	15	43 (24+19)
$\rho 6$	org.pingchuan.dingwork	Telephony	9,463	88	85 (52+33)
$\rho 7$	com.customer.taoshijie.com	Shopping	7,216	68	75 (28+47)
$\rho 8$	com.thestore.main	Shopping	10,483	172	75 (36+39)
$\rho 9$	com.zlianjie.coolwifi	System Tool	5,596	55	101 (52+49)
$\rho 10$	com.dp.android.elong	Travel	22,524	351	61 (36+25)

Table F4 Test results of A3E and OA3E to trigger API invocations in $API1$ and $API2$ of the 10 apps. "S", "D" and "T" mean the time costs of static analysis, dynamic exploration and the total test time respectively. "APIs Trigger" means the number of target API invocations that be triggered.

APP Id	A3E				A^+ Size	OA3E				Time Compare OA3E/A3E
	Time Cost (minutes)			APIs Trigger		Time Cost (minutes)			APIs Trigger	
	S	D	T			S	D	T		
$\rho 1$	7	176	183	19	47	23	49	72	19	39.34%
$\rho 2$	5	133	138	136	54	18	52	70	134	50.72%
$\rho 3$	3	84	87	57	32	11	26	37	57	42.53%
$\rho 4$	6	163	169	80	45	12	73	85	79	50.30%
$\rho 5$	1	25	26	25	13	4	15	19	25	73.08%
$\rho 6$	5	145	150	46	35	19	30	49	43	32.67%
$\rho 7$	3	75	78	41	37	8	28	36	41	46.15%
$\rho 8$	8	228	236	50	53	19	51	70	50	29.66%
$\rho 9$	2	42	44	59	39	10	26	36	58	81.82%
$\rho 10$	11	317	328	39	78	28	70	98	39	29.88%

nearly same GUI events sequence for the GUI elements in G_a^+ for each activity $a \in A^+$.

In order to better demonstrate the experimental results, we randomly selected 10 apps from these 100 apps and showed their test results in Table F3 and F4 (We did not lists more apps due to limited space, and there is no need for this). Table F3 lists the basic information of these 10 apps including their package names, the categories they belong to, sizes, activities numbers they have and the numbers of the target API invocations of $API1$ and $API2$. The "App Id" shown in the first column is what we used to identify each app. Table F4 shows the test results of A3E and OA3E for these 10 apps, including test time (Static analysis time, dynamic exploration time and the total time), the numbers of API invocations that be triggered of $API1$ and $API2$. In the last column, the total test time of OA3E for each app is compared with A3E by a ratio of OA3E to A3E.

It can be seen from Table F3 and F4 that both A3E and OA3E cannot achieve the full coverage of the target API invocations. It is caused by many reasons, such as the target API invocations in the other three components except for activity do not depend on the GUI events, or the trigger of some API invocations depends on some particular conditions that cannot be satisfied, such as the arrival of a specific time. As the main concern of this paper is how to make a GUI exploration tool to avoid wasting time on exercising the activities and GUI elements that cannot lead to the target API invocations, therefore how to make a GUI exploration tool to trigger as many API invocations as possible is not within the scope of our consideration. This will be left as our future improvement work on OA3E.

Appendix F.3 Efficiency Analysis of OA3E

From Table F4, it can be seen that the ratios of total test time of OA3E to A3E vary from 29.88% to 81.82%. This is a big span and it means that OA3E is not very efficient under any circumstances. This section we will discuss the factors that affect OA3E's efficiency.

Combine Table F3 with Table F4, it can be seen that OA3E can obviously reduce the time costs for apps that contain more activities inside. For instance, in Table F3, apps $\rho 8$ and $\rho 10$ both have large activities numbers. But we can see from Table F4 that both the ratios of total time costs of OA3E to A3E do not exceed 30%. On the contrary, if an app has a very small activities number and the size of A^+ is very near to the activities number, then OA3E has no obvious advantages. For instance, it can be seen from Table F3 that apps $\rho 5$ and $\rho 9$ both have very few activities. Table F4 shows that their

sizes of A^+ are very near to the activities numbers and both the ratios of total time costs of OA3E to A3E exceed 70%.

We can see clearly another phenomenon that, with similar activities numbers, more target API invocations means more activities need to be exercised and more time cost for each app. For instance, it can be seen from Table F3 that apps $\rho 2$ and $\rho 6$ have very similar activities numbers, but the number of target API invocations in $\rho 2$ is more than $\rho 6$. Accordingly, $\rho 2$ has a bigger size of A^+ and more total test time than $\rho 6$. On the contrary, the activities numbers of apps $\rho 3$ and $\rho 7$ are also very similar, but the numbers of target API invocations in $\rho 3$ and $\rho 7$ are similar as well. Accordingly, their total test time are very similar.

To better test the relationship between target APIs number, size of A^+ and the total test time, we did our second experiment. In this experiment, we chose app $\rho 10$ in Table F3 which has a large activities number as the experimental object. The target APIs varied from $API1$ to API5 in the form of accumulation (i.e., $API1$, $API1 + API2$, $API1 + API2 + API3$, $\cdots$). The test results are shown in Figure F1. From the test results, we can see that with the target API sets increased from $API1$ to $API1$-5 (i.e., $API1 + \cdots + API5$), the target APIs number, size of A^+ and the total test time of $\rho 10$ are increasing. When OA3E only used the APIs in $API1$ as target APIs, the total test time is 66 minutes which is only 20.12% of A3E's ($\frac{66}{328}$). However, when OA3E used all the APIs in the API sets from $API1$ to $API5$ as target APIs, the total test time is 257 minutes which is 78.35% of A3E's ($\frac{257}{328}$). It can be concluded 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 (so OA3E can filter out more activities and the size of A^+ is small), then OA3E would cost less test time and be more efficient than A3E. This experiment also gives an important tip to the users of OA3E that, in order to make full use of OA3E's advantages in triggering the target API invocations, the input target APIs must be reduced to a minimum set, without affecting the observation of the target program behaviors that the users concerned about.

Appendix G Discussion

After the introduction of the experiments we had done on OA3E, we then introduce the possible usability of OA3E, its limitation and the future work needs to be done for it.

Appendix G.1 Usability of OA3E

We can think of two important application scenarios that OA3E can be used in, but it is not limited to these scenarios:

(1) OA3E can be used to trigger sensitive behaviors of the apps, which is very useful in dynamic detecting of Android malware. In the process of dynamic detecting Android malware, it is essential to trigger as many sensitive behaviors as possible of the apps under test during their actual running process, so that the detection tools can capture and analyze these behaviors to judge whether the apps under test are malicious apps. When we define all of the sensitive APIs which the detection tools care about in the target API file of OA3E, OA3E would automatically drive the apps to trigger these sensitive API invocations in them to expose their sensitive behaviors.

(2) OA3E can be used to trigger the possible vulnerabilities of the apps, which is useful for the security analyzers to find and patch the security vulnerabilities of the apps. For example, in order to detect whether an app has privacy leakage vulnerabilities [11] [12], it is often needed to analyze whether the data that be transmitted out of the phone contains privacy-related data. But how to trigger app's behavior of reading the sensitive data and sending this data outside the phone is a big problem [11]. In this case, OA3E can be used to trigger these behaviors by defining the specific target APIs that related to reading and sending sensitive data (such as the APIs sets $API1$ and $API2$ described in Section Appendix F.1) in the target API file. Besides, Ravi Bhoraskar [14], et al introduced in their paper [14] that how to find the vulnerabilities in third-party components by the method of triggering the target API invocations, which can be accomplished with the help of OA3E as well.

Appendix G.2 Limitation and Future Work

OA3E can be used to guide an app to trigger the target API invocations that the testers interested in, and its effectiveness and efficiency had been verified through experiments. However, it still has some limitations that need to be improved:

(1) Limitation of static analysis. Both A3E and OA3E use the method of static analysis, but sometimes the static analysis method is not accurate enough, which will affects OA3E's results of Activity Filtering and GUI Filtering. Two typical problems are—the callback mechanism of Android framework affects the completeness of control flow graph of an app [5] and the reflection and dynamic loading mechanism can be hardly analyzed by the method of static analysis [15]. For the former problem, OA3E refers to the solution of FlowDroid [16], which models life-cycle related callbacks of Android by means of a manually created list which lists all the callbacks and their registration methods in Android framework. Besides, the results generated by EdgeMiner [5] are incorporated by OA3E to achieve a more complete control flow graph (which had been introduced in Section Appendix C.2). For the latter problem, OA3E takes measures of defining the APIs that related to reflection and dynamic loading mechanism (e.g., "invoke", "loadClass", etc.) in the target API file of OA3E to trigger the behaviors of reflection and dynamic loading of an app. Although through these methods, OA3E had achieved good results (look at the ratio 98.63% described in Section Appendix F.2), but OA3E's static analysis method still needs to be improved to achieve a higher performance.

(2) The GUI element features that OA3E uses are not strong enough. OA3E uses three features to identify a GUI element—view ID, displayed text and GUI element type. Some combinations of them can uniquely identify a GUI element which had been introduced in Section Appendix D.4. However, through the experiments, we found that too many GUI

elements cannot be uniquely identified by the combinations of these features, such as the GUI elements that showed as images. It will affect the effect of GUI Filtering, because the more GUI elements that cannot be found unique features in an activity a, the smaller size of F_a^+ and F_a^-, and the poorer effect of GUI Filtering.

(3) OA3E cannot find the relationship between user input and execution path of the program. The specific inputs affect the flow of the program [17], but both A3E and OA3E cannot judge the right inputs that lead to a specific execution path of the program. For example, a radio button should be set to a specified value to lead the program to trigger a target API invocation. But OA3E cannot judge which value should be set to the radio button and it just sets a random value to it. Some researchers used the method of symbolic execution to solve this problem [17] [18], but according to our survey and study, we concluded that symbolic execution is far from a mature technology to be used in Android app analysis and it is hard to strike a balance between the real valid inputs and the large time costs of symbolic execution. Therefore, whether symbolic execution or any other methods should be used to generate the valid inputs for an app to trigger the target API invocations will be left as our future work on OA3E.

In the future work, we will take measures to improve OA3E's accuracy of static analysis. New features will be applied to identifying a GUI element. Some heuristic mechanisms will be applied to find a GUI element and trigger its GUI event. Besides, we will take measures to generate valid inputs to make OA3E to be more accurate to trigger the target API invocations.

References

1 Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of android apps. ACM SIGPLAN Notices, 2013, 48(10): 641-660

2 Fuchs A P, Chaudhuri A, Foster J S. Scandroid: Automated security certification of android applications. Technical Report CS-TR-4991. 2009

3 Android Developers. App Components. http://developer.android.com/guide/components/index.html

4 SourceForge. WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page

5 Cao Y, Fratantonio Y, Bianchi A, et al. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In: Proceedings of the 22th Network and Distributed System Security Symposium, San Diego, CA, USA, 2015

6 Tarjan R. Depth-first search and linear graph algorithms. SIAM journal on computing, 1972, 1(2): 146-160

7 Wang L. Encyclopedia of Systems Biology. New York: Springer, 2013. 574-574

8 Android Developers. Input Events. http://developer.android.com/guide/topics/ui/ui-events.html

9 Lhotk O, Hendren L. Scaling Java points-to analysis using Spark. In: Proceedings of the 12th International Conference on Compiler Construction, Warsaw, Poland, 2003. 153-169

10 Jeon J, Foster J S. Troyd: Integration Testing for Android. Technical Report CS-TR-5013. 2012

11 Gibler C, Crussell J, Erickson J, et al. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: Proceedings of the 5th International Conference on Trust and Trustworthy Computing, Pittsburgh, PA, USA, 2012. 291-307

12 Zhou Y, Zhang X, Jiang X, et al. Taming information-stealing smartphone applications (on android). In: Proceedings of the 4th International Conference on Trust and Trustworthy Computing, Pittsburgh, PA, USA, 2011. 93-107

13 Xposed Module Repository. Xposed Installer. http://repo.xposed.info/module/de.robv.android.xposed.installer

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

15 Zhauniarovich Y, Ahmad M, Gadyatskaya O, et al. StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 2015. 37-48

16 Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Notices, 2014, 49(6): 259-269

17 Yang Z, Yang M, Zhang Y, et al. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, Berlin, Germany, 2013. 1043-1054

18 Schutte J, Fedler R, Titze D. Condroid: Targeted dynamic analysis of android applications. In: Proceedings of the 29th International Conference on Advanced Information Networking and Applications, Gwangju, Korea, 2015. 571-578