2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

FragDroid: Automated User Interface Interaction with Activity and Fragment Analysis in Android Applications

Jia Chen[†], Ge Han[†], Shanqing Guo*[†‡] and Wenrui Diao[§]
[†]School of Computer Science and Technology, Shandong University
Email: chenjia@mail.sdu.edu.cn, hangehg@126.com
[‡]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education,
Shandong University, Jinan 250100, China
Email: guoshanqing@sdu.edu.cn
[§]Jinan University
Email: diaowenrui@link.cuhk.edu.hk

***Abstract*—Recent years have witnessed the enormous growth of Android phones in the consumer market. On the other hand, as the most popular mobile platform, Android also attracts lots of attackers' attention. As a result, more and more Android malicious apps appear in the wild, which poses a serious threat to user's security and privacy. To such massive volume of Android malware, automated UI testing techniques have become the mainstream solutions because of the detection efficiency and accuracy. However, all existing UI testing techniques treat the Activity as the basic unit of UI interactions and cannot carry out a fine-grained analysis for Fragments. Due to the lack of Fragment-level analysis, the path coverage is usually quite limited.**

To fill this gap, in this paper, we propose FragDroid, a novel automated UI testing framework supporting both Activity and Fragment analysis. To achieve the Fragment-level testing, we design the Activity & Fragment Transition Model (AFTM) to simulate the internal interactions of an app, and ATFM could be utilized to generate test cases automatically through UI interactions. With the assist of AFTM, FragDroid achieves accessing most Activities and Fragments contained in the app along with the capability of detecting arbitrary API calls. We implemented a prototype of FragDroid and evaluated it on 15 popular apps. The results show FragDroid successfully covered 66% Fragments and the corresponding API calls of testing apps. Also, the traditional approaches have to miss at least 9.6% of API calls invoked in Fragments.

I. Introduction

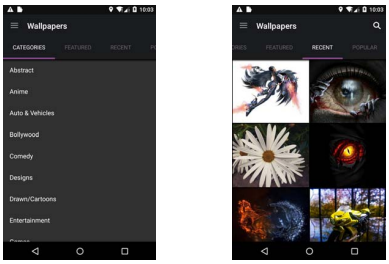
The powerful functionalities of smartphones are primarily supported by diverse mobile applications (*apps* for short). As the most popular mobile platform, Android provides millions of apps for users. There are over 1.5 million applications and over 50 billion downloads on Google Play - Android's official app store. Android's popularity also attracts attackers' attention. More and more malicious apps appear in the wild, which poses a serious threat to user's security and privacy. To such massive volume of Android malware, automated UI testing techniques have become the mainstream solutions because they could achieve the balance of detection efficiency and accuracy. The original approach of UI testing is to inject random test cases into a running app to explore UI states as many as possible. The most representative tool is Monkey [1] which is provided by Google. After that, as an enhancement, the record and replay (R&R) test technique was proposed [2], [3]. Such technique could record the UI events triggered by human testers and translate them to scripts. The scripts can then be executed on other devices to drive the app running through replaying the recorded UI events. The R&R test technique could reproduce the test cases easily, but its cost is quite expensive in the input collection and maintenance.

More recently, model-based testing (MBT) technique was proposed, which injects test cases into an app aligning with a specific model. MBT usually contains two phases – model generation and dynamic testing. The challenge of MBT is how to generate an effective model with high path coverage rate. Automatic model generation based on the source code is usually used in white-box tests, while dynamic slicing at runtime techniques is usually used in black-box tests. For dynamic slicing, the model generated by a slicer based on UI states guides the app to trigger events, and the shortcomings include the difficulty of backtrack and the lack of context.

Activities are the fundamental building blocks of Android apps. Existing Android UI test tools based on MBT mainly use the Activity as the basic unit to distinguish different UI states. However, since Android 3.0, the Fragment has been introduced into Android, and it could be treated as sub-Activity or mini-Activity. The usage of Fragments has become popular after Android 5.0 because of its high efficiency and low consumption in the UI switching. To demonstrate its popularity, we conducted a study on 217 top downloading apps from Google Play, and it shows that nearly 91% of these apps use Fragments. At the same time, none of existing MBT techniques could handle the widespread Fragment components properly, and the challenges derive from multiple aspects.

Challenge 1: An Activity could host Fragments, and nearly all existing MBT techniques take Activity as a fixed UI state. The existing techniques neglect the reachable UI states caused

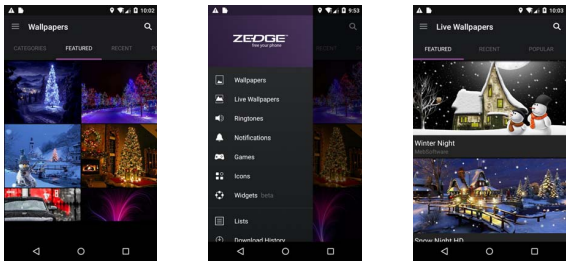
2158-3927/18/$31.00 Â©2018 IEEE
DOI 10.1109/DSN.2018.00049



(a) Fragment of Category tab

(b) Fragment of Recent tab

Fig. 1: Fragment Transformation



(a) Fragment of wallpapers

(b) Slide menu

(c) Fragment of live wallpapers

Fig. 2: Fragment switching through hidden slide menu

by Fragment transformations and miss the logic functions contained in them. Although random input tests, such as Monkey, can occasionally reach these Fragments, they are not programmable and cannot be controlled accurately. To achieve the Fragment-level analysis, the transitions between Activities and Fragments and the functions contained in Fragments need to be analyzed. Following this requirement, it has to be considered that a transformation of Fragments inside an Activity may lead a switching of UI states. As an example shown in Figure 1, the clicking on the menu will trigger a Fragment transformation (from Figure 1(a) to Figure 1(b)), which further occurs the switching of UI states. In this case, we could find the object of the rest testing operations is changed while the Activity is not. If the testing tools ignore Fragment transformations, the proportion of total reachable UI states will stay at a low level.

Challenge 2: The approaches to switching Fragment components could be invisible or hidden in UI, so the existing MBT techniques may fail to uncover the relationships of Fragments. Figure 2 illustrates this situation, in which an app uses the navigation drawer design. Figure 2(a) and Figure 2(c) show two Fragments in the same Activity, and the slide menu in Figure 2(b) is the only bridge between them. Also, the menu is hidden and only can be seen by clicking the left-top icon or sliding from left to right. Most existing techniques neglect such kind of switching relationships and the corresponding UI states.

Our Work. To address the challenges caused by Fragment components, we propose FragDroid, a novel automated Android UI testing technique supporting both Activity and Fragment analysis. FragDroid takes the interactions of Activities and Fragments into consideration during the test. It could detect and invoke both Activities and Fragments to explore all reachable UI states.

In the design of FragDroid, we define an Activity & Fragment Transition Model (AFTM) which could be evolutionarily updated to store the possible transitions between Activities and Fragments. AFTM considers the dependencies among UI elements including Activities, Fragments, and widgets. To address the feature of hidden switching of Fragments, we use the Java reflection mechanism to switch UI states forcedly. Also, the design of FragDroid synthesizes the idea of ripper techniques and automation frameworks/APIs (AF/A) [4], [5], [6] (providing the high-level syntax for generating test cases). Given an APK file, FragDroid extracts the dependency information first. Then this app is installed on a customized Android device for dynamic execution. FragDroid generates proper test case scripts base on AF/A, and test cases will trigger different UI events. At the same time, FragDroid monitors and analyzes the runtime information. Once the UI state (on the Fragment level) changes, the AFTM and the sequence of test cases will be updated until all possible UI states have been explored.

Contributions. The main contributions of this paper are:

- We proposed Fragdroid, the first Android automated UI testing framework supporting both Activity and Fragment analysis. We also implemented a full-feature prototype of FragDroid.
- We analyzed 217 popular apps from different categories and revealed that up to 91% of apps are developed with Fragments. We evaluated FragDroid on 15 of them, and the average coverage is 66% for Fragments and 71.94% for Activities. It demonstrates that FragDroid achieves a satisfactory coverage and reaches a recommendable performance.
- We applied Fragdroid to discover the relation between the invocations of sensitive APIs and UI elements (including Activities and Fragments), which is helpful for detecting malicious code, bug, etc. In the experiment, 46 sensitive APIs, like obtaining locations and accessing storage, were found through deploying FragDroid on 15 selected apps. The result shows that the API invocations associated with Fragments account for 49% of the total invocations. The traditional approaches based on Activity have to miss at least 9.6% of API calls invoked in Fragments.

Roadmap. The rest of this paper is organized as follows. Section II gives the necessary background of Activity and Fragment. In Section III, we present the high-level design of FragDroid. Section IV provides the definition and initialization the Activity & Fragment Transition Model. Section V and Section VI describe the process of dependency extraction and evolutionary test case generation respectively. The experimen-

tal results are presented in Section VII. Section VIII discusses the limitation of our framework. Section IX summarizes the related works, and Section X concludes this paper.

II. Background

In this section, we briefly introduce the Activity and Fragment components in Android.

A. Activity and Fragment in Android

The Activity class is the most common component of an Android app, and the way in which Activities are launched and put together plays a fundamental part in application model [7]. It serves as the entry point to interact with users and provides a window where the app draws its UI.

The Fragment was introduced in Android 3.0 to facilitate the app development and better the user experience [8]. It supports more dynamic and flexible UI designs on large screens. A Fragment could be treated as a modular section of an Activity (or mini-Activity), which has its own lifecycle, receives its own input events. By dividing the layout of an Activity into Fragments, developers become able to modify the Activity's appearance at runtime [9].

B. The Relationship of Activity and Fragment

In simple terms, the Activity and Fragment are both the fundamental building blocks of Android apps. A Fragment could be treated as a sub-Activity or mini-Activity. It could be added to an Activity or removed from an Activity at runtime.

Usually, a Fragment contributes a portion of UI to the host Activity, which is embedded as a part of the Activity's overall view hierarchy. Developers can attach the Fragment layouts they want to inflate by implementing the `onCreateView()` callback method. There are two ways to add a Fragment to the Activity layout: (1) declare the Fragment inside an Activity's layout file; (2) add the Fragment into an existing ViewGroup programmatically. The `FragmentTransaction` class [10] provides the APIs for performing a set of Fragment operations, including adding, removing, and replacing a Fragment. The code snippet listed in Figure 3 shows how to add a Fragment to an Activity at runtime.

In addition, developers can combine multiple Fragments in a single Activity to build a multi-pane UI and reuse one Fragment across multiple Activities. Since a Fragment can get Context instance from its host Activity, it can execute almost all actions like an Activity, such as starting a new Activity, obtaining privileges, accessing sensitive information, and so forth.

III. System Overview

In this paper, we propose FragDroid, an automated Android GUI testing framework supporting Fragments. It can trigger nearly all Fragments and Activities during dynamic analysis to achieve a high path coverage. As illustrated in Figure 4, FragDroid contains two main phases: *Static Information Extraction* and *Evolutionary Test Case Generation*. Here we give a brief overview.

Static Information Extraction. Based on the static code analysis, this phase aims to collect the necessary information to facilitate the subsequent evolutionary test case generation phase.

- The primary information collected is the Activity & Fragment Transition Model (AFTM) which is a finite state model simulating the internal interactions among Activities and Fragments. This model is extracted from the smali code[1] of the target app. The formal definition of AFTM will be given in Section IV-A.
- Also, some meta-data used for the evolutionary testing will be collected, like the number of Activities and Fragments, dependencies among UI controls, etc. Especially, we provide a JSON file that records all view components and the locations they appear.

Evolutionary Test Case Generation. In this phase, dynamic test cases are generated with the data in AFTM. Note that, AFTM is a dynamic model, and it will be updated continuously until all nodes have been visited.

In the beginning, the *queue generation* module traverses the initial AFTM by breadth-first search. Every newly discovered node (Activity or Fragment) in the AFTM will trigger that a new item will be pushed to the queue. This item contains the information of the transition from the entry node to the discovered node, like the way to reaching particular interface and a series of UI events during the transition. After that, the item in the queue will be put to the *test case generation* module for generating a test program by Robotium. With the meta-data collected during static analysis, the test program will be created and installed to the phone automatically.

After running a test program, the *UI driving* module analyzes the current UI state on the Fragment level. Three tasks are involved: (1) identifying the current Activity and Fragment based on the previously extracted resource dependency; (2) triggering all clickable widgets one by one; (3) analyzing the new UI state after clicking operations and updating the AFTM (if a new transition relationship appears). The above operations will not stop until all nodes have been visited.

IV. Activity & Fragment Transition Model

To facilitate the execution of our dynamic analysis framework, we propose the Activity & Fragment Transition Model (AFTM). AFTM is a finite state model extracted from an Android app. It contains all working Activities, Fragments, and the event-driven transitions among them. Working Activities and Fragments mean they are not isolated and could interact with users. Essentially, such model simulates the internal structure of an app and could be treated as a map for dynamic analysis.

In this section, we give the formal definition of AFTM and describe how to initialize this model from an app as well as other associated tasks during static analysis.

[1]After reversing an APK file, we could get the corresponding smali code, like the decompilation code from binary code.

```java
// Get an instance of FragmentTransaction from an Activity
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

// Use the add() method to add a Fragment
// ExampleFragment class extends from android.app.Fragment
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);

// If we want to replace the current Fragment with ExampleFragment, use:
// fragmentTransaction.replace(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

Fig. 3: Code example: Add a Fragment to an Activity

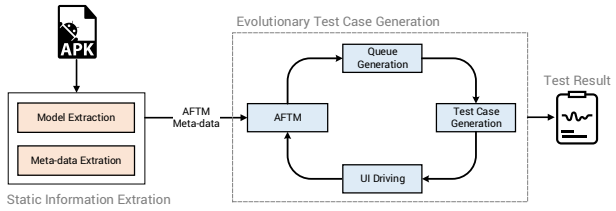


Fig. 4: Overview of FragDroid

A. Definition of AFTM

Definition 1. The AFTM of an app is a tuple $< A, F, E >$, where

- A is a finite set of Activities that can switch from/to other elements (Activities or Fragments) in the call graph of the app. $A0$ is the entry Activity, and so on, for $A1$, $A2$, $A3$, and ....
- F is a finite set of Fragments that can switch from/to other elements in the call graph. Similarly, we have $F0$, $F1$, $F2$, and ....
- E is a finite set of transition relationships among Activities and Fragments. There are three basic relationships:
  1) $E1 : A \rightarrow A$ (outer): From an Activity to another Activity directly. Since there doesn't exist $A \rightarrow A$ (inner), we will use $A \rightarrow A$ to represent it.
  2) $E2 : A \rightarrow F$ (inner): From an Activity to its own Fragments. We will use $A \rightarrow F_i$ to represent it.
  3) $E3 : F \rightarrow F$ (inner): From a Fragment to another Fragment. Both of them belong to one Activity. We will use $F \rightarrow F_i$ to represent it.

Note that, in fact, there are seven types of transition: $A \rightarrow A$, $A \rightarrow F_i$, $F \rightarrow F_i$, $A \rightarrow F_o$, $F \rightarrow A_i$, $F \rightarrow A_o$, and $F \rightarrow F_o$, where F_i (A_i) stands for an internal Fragment (Activity), and F_o (A_o) stands for an external Fragment (Activity). Finally, we merge them into three situations as mention above. The other four situations are ignored. First, we ignore the edge $F \rightarrow A_i$ because this transition must go through its host Activity. Second, all edges starting from a

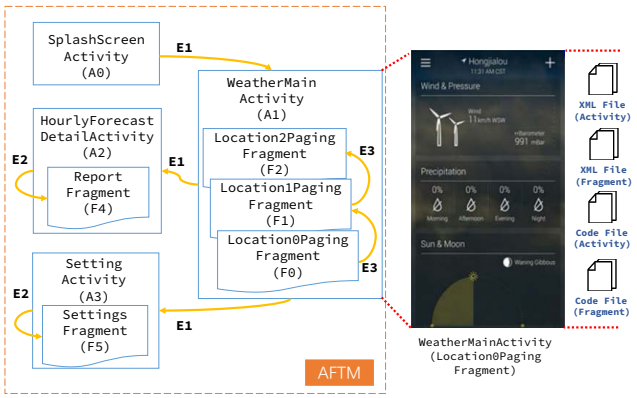


Fig. 5: Example of AFTM

Fragment can be regarded as starting from its host Activity. Therefore, $F \rightarrow A_o$ and $F \rightarrow F_o$ can be considered as $A \rightarrow A_o$ and $A \rightarrow F_o$ respectively. In Definition 1, $A \rightarrow A_o$ is equal to $A \rightarrow A$. Third, $A \rightarrow F_o$ can be split into $A \rightarrow A$ and $A \rightarrow F_i$.

After merging, all transitions among Activities and Fragments can be expressed by the three basic edges. In the paper, we use $E1$ to represent $A \rightarrow A$, $E2$ for $A \rightarrow F_i$, and $E3$ for $F \rightarrow F_i$. In Figure 5, we give an abstract view of AFTM of an app, and we could find the three basic edges ($E1$, $E2$, and $E3$) could cover all situations.

B. Initialization of AFTM

In this subsection, we introduce how to generate the AFTM from an app.

1) Decompile APK: We use Apktool [11] to decompile the target APK file to get the smali code and its AndroidManifest.xml file. The initialization construction of AFTM is based on them. Also, in this step, we further convert the smali code to the corresponding Java code through jd-core [12] for the last step – transition edge calculation.

2) Get the Effective Activities and Fragments: In order to ensure the accuracy of AFTM, we must remove the invalid Activities and Fragments, which have no interactions with other

UI elements. Invalid Activities include the Activities involved in intermediate classes as well as isolated Activities. Through analyzing the manifest file, we can get a list of all declared Activities. Also, this list does not contain the Activities in intermediate classes, so the interference of intermediate classes can be solved. Then, we filter out the isolated Activities. If an Activity is considered as a node, the interaction between two nodes is an edge. It is clear that when completing the classification and acquisition of all the edges, the nodes not linked by any edge are isolated and should be removed.

To Fragments, through scanning all the decompiled smali code files, we can find some files that inherit from the Fragment class. These files are the subclasses of Fragment, and we save the class names to a list. Next, we scan all smali files again to find out all derived classes that inherit from these subclasses of Fragment and add the newly discovered ones to our collection. Note that this collection still needs to be filtered. Here we assume that we have obtained a valid list of Activities. Then we collaborate the effective Activity classes with the Fragments in the collection to observe whether there exists a statement of the Fragment. If the statement could be found, we consider the corresponding Fragment is an effective Fragment. Following this approach until all Fragments in the collection are checked, we will get an updated effective collection in the end.

3) Get the Transition Edges: We treat the Activities and Fragments found in the static analysis as nodes, so the transition relationships among them are edges. There are seven types of edges in practice. Note that, as mentioned before, the seven types will be merged to three basic types: $A \rightarrow A$, $A \rightarrow F_i$, and $F \rightarrow F_i$. We designed an Algorithm to emulate those edges and generate the AFTM graph, as listed in Algorithm 1.

As shown in Algorithm 1, to the edge $A \rightarrow A$, we should analyze the file of every Activity. Whenever we want to switch to a new Activity, we need to create an Intent object with the information of this converted Activity and execute the `startActivity()` method. If there exists Java code like the form of `new Intent(Class A0, Class A1)` or `setClass(Class A0, Class A1)` in $A0$ class, the second parameter (`Class A1`) could be used as the information of the Activity converted to because it indicates the name of the new Activity. Then we will add this switching relationship $A0 \rightarrow A1$ to the list. To the code `new Intent(String action)` or `setAction(String action)`, it is a different case because the String type of parameter indicates the information of the Action in AndroidManifest.xml file. Therefore, we have to find the corresponding statement in AndroidManifest.xml, determine the Activity it belongs to, and add the corresponding edge and nodes to the graph.

To the edge with Fragment as the end node, say $A \rightarrow F_i$ and $F \rightarrow F_i$, we look for the code of instances of Fragment $F1$ in Activity $A0$'s class file or Fragment $F0$'s class file. Then If $F1$ belongs to $A0$ or $F0$ and $F1$ belong to the same Activity, these edges and nodes will be added to the graph.

Algorithm 1 Generate AFTM Graph

Input:
Current Activity java file called A0.java
Current Fragment java file called F0.java
AndroidManifest.xml
Output:
The AFTM Graph $G = (V, E)$
function GETEDGEATOAORATOF()
for all lines in $A0.java$ **do**
if contains $setClass(Class\ A0, Class\ A1)$ or $new\ Intent(Class\ A0, Class\ A1)$ **then**
$V = V \cup A0$;
$V = V \cup A1$;
$E = E \cup (A0 \rightarrow A1)$;
end if
if contains $new\ Intent(String\ action)$ or $setClass(String\ action)$ **then**
if $find\ A1\ in\ AndroidManifest.xml\ by\ action$ **then**
$V = V \cup A0$;
$V = V \cup A1$;
$E = E \cup (A0 \rightarrow A1)$;
end if
end if
if contains $new\ F1()$ or $instanceof(F1)$ or $F1.newInstance()$ **then**
if $F1 \in A0$ **then**
$V = V \cup A0$;
$V = V \cup F1$;
$E = E \cup (A0 \rightarrow F1)$;
end if
end if
end for
end function
function GETEDGEFTOF()
for all lines in $F0.java$ **do**
if contains $new\ F1()$ or $instanceof(F1)$ or $F1.newInstance()$ **then**
if $F_0, F_1 \in A$ **then**
$V = V \cup F0$;
$V = V \cup F1$;
$E = E \cup (F0 \rightarrow F1)$;
end if
end if
end for
end function

V. Static Information Extraction

FragDroid needs the dependency information to identify states and analyze relationships in apps. Such information is provided as knowledge for the subsequent evolutionary test case generation phase.

In the previous research, TrimDroid [13] explains widget, handler, and Activity dependency on the level of Activity. As

```
// Create intent for next Activity
Intent intent = new Intent(Context,
    SecondActivity.class);

// Start the Activity from Activity
startActivity(intent);

// Start the Activity from Fragment
getActivity().startActivity(intent);
```

Fig. 6: Code example: Start next Activity

the Activity dependency in TrimDroid, FragDroid extracts the dependency relationships among Activities and extends the scope to the Fragment level as well as the dependency relationships between Activities and Fragments. Besides, resource dependency and input dependency are the essential parts of the collected meta-data. The Activity & Fragment dependency specifies the Activities and Fragments impacted by the behaviors of another Activity or Fragment. The resource dependency helps the UI driving component to quickly distinguish which Activity or Fragment the current UI belongs to through source-IDs. The Input dependency is built up manually to provide proper inputs for widgets like `EditText` so that the test cases could reach more states.

A. Activity & Fragment Dependency

It is known that the switching between Activities could be proceeded by Intent. Usually, an Intent contains many serialized data some of which are concerned with the widget state. Likewise, in a Fragment, the Intent can be used for the transition from a Fragment to an Activity. This can be implemented by calling the Context of host Activity in a Fragment via function `getActivity()`, as shown in the code snippet in Figure 6. In some cases, a Fragment shares events with its host Activity. One way to achieve that is to define a callback interface inside the Fragment and require the host Activity to implement.

Furthermore, a Fragment may be used in one or more Activities, which means `ExampleFragment` in Figure 3 can appear in any Activity which involves Fragment components. In reverse, an Activity is able to hold more than one Fragment. The code in Figure 3 shows how to add a Fragment to an Activity, in which `ExampleFragment` can be replaced with any subclass of `android.app.Fragment`. To the example shown in Figure 1, in order to discover more possible test paths, it would be helpful if we identify the Fragments of CATEGORY and RECENT tabs in the current Activity. For such a purpose, Algorithm 2 describes the procedure of identifying the dependency among Activities and Fragments. First, FragDroid gets all used classes from each Activity and its inner classes (like `ExampleActivity$1.class`) and then analyzes the inheritance chain of each used class. If there exists class `android.app.Fragment` or class `android.support.v4.app.Fragment` in the inheritance chain, this class is a derived class of Fragment. Then this Activity and class are the dependency of the Activity and Fragment.

Algorithm 2 Activity & Fragment Dependency

Input: $a \in Activity$
Output: The Relationship of Activity & Fragment $R = (A, F)$

```
R ← ∅
FClass ← "android.app.Fragment"
sFClass ← "android.support.v4.app.Fragment"
for all a in Activity do
    allClass ← getInnerClass(a)
    for all aClass in allClass do
        Classes ← getUsedClass(aClass)
        for all Class in Classes do
            classChain ← getSuperChain(Class)
            if FClass ∈ classChain || sFClass ∈ classChain then
                R = R ∪ {a, Class}
                break
            end if
        end for
    end for
end for
```

B. Resource Dependency

Android UI is constituted with numerous UI components. A UI component in the present UI state always belongs to some certain Activity. In existing research, several Activity-level analysis tools design their UI models by analyzing the layout files and code of Activities [13]. However, the introduction of Fragment makes the Activity-based UI models incomplete, because a widget in current UI state may belong to an Activity or a Fragment. For instance, in Figure 1(a), the listener of the tab marked as "CATEGORIES" belongs to an Activity, but the list below is implemented in a Fragment. It means the code of different widgets in the same UI may be implemented in different files.

The static analysis phase extracts the resource dependency to match widgets to their host Activities and Fragments. In Android, a unique number (resource-ID) is used to identify a resource. As shown in Algorithm 3, the dependency information is collected by discovering the resource-IDs that repeatedly appear in both layout and resource files. Function `getID()` fetches a widget's resource-ID, and function `getAID()` / `getFID()` outputs a list of resource-IDs contained in an Activity or Fragment. At the same time, all non-interaction widgets not declared in code file are ruled out.

C. Input Dependency

Android apps usually require users to enter some information to complete a specific function. Different input values usually lead to different outcomes. For example, in the login Activity of an app, only the correct account information can

Algorithm 3 Resource Dependency

Input: $a \in Activity$, $f \in Fragments$, $L \in Layouts$
Output: The AFRM Model $M = (A, F, RID)$

```
M ← ∅
for all l in Layouts do
    for all w(idge) in l do
        id ← getID(w)
        isFind = false
        for all a in Activity do
            aID ← getAID(a)
            if id ∈ aID && l ∈ a then
                M = M ∪ {a, null, w}
                isFind = true
                break
            end if
        end for
        if !isFind then
            for all f in Fragment do
                fID ← getFID(f)
                if id ∈ fID && l ∈ f then
                    M = M ∪ {null, f, w}
                    break
                end if
            end for
        end if
    end for
end for
```

let the test process move on. Without a successful login, most of the subsequent Activities will not be reached. As another example, a search box in *TheWeatherChannel* requires inputting the name of an existing place for checking weather information. If a test tool inputs random string like "abc", this app may report an error or give the null result, and the test cannot continue.

There are many studies on improving the input for dynamic analysis. TrimDroid [13] considers the input relationship of different widgets. Chen et al. [14] propose a simple way to generate input according to the state of a widget and its context. Dynodroid [15] cuts down impossible input sequence combinations to decrease the number of generated test cases. FragDroid utilizes some techniques of these works to ensure that it could generate inputs as accurate as possible. Moreover, FragDroid introduces a new input interface which is a file containing resource-IDs of all input widgets (like EditText, CheckBox, and so on). Regarding this file, analysts can manually fill the input fields with correct values in advance, then FragDroid will use these values with a preference during tests.

VI. Evolutionary Test Case Generation

This section discusses how the AFTM is iteratively updated. It is a kind of dynamic process that the output of the previous test is fed back to AFTM, and new test cases will be generated according to the update of AFTM. This process ends when all nodes in AFTM model are accessed and all test cases have been executed, and no new node is added to AFTM.

A. UI Driving and AFTM Update

FragDroid could drive the execution of a test case automatically and reach the target interface set in the case. There are three methods for FragDroid to reach a certain interface:

- We use the command `am start -n <COMPONENT> -a android.intent.action.MAIN -c android.intent.category.LAUNCHER` to launch an app by Android Debug Bridge (ADB) [16], in which `<COMPONENT>` is the entry Activity.
- We translate the operation series of test cases and stored them as a test script, then package them into the target Android app by Ant [17] and install this app to the testing phone. Finally, we use the ADB command `am instrument -w <TestPackageName> android.test.InstrumentationTestRunner` to run this app, in which `<TestPackageName>` is the package name of this app.
- During static analysis, we modify AndroidManifest.xml by adding the attribute `<action android:name= "android.intent.action.MAIN"/>` for every Activity and use the ADB command `am start -n <COMPONENT>` to forcibly start an Activity which FragDroid cannot visit by normal methods in the first phase of dynamic testing, in which `<COMPONENT>` is the target Activity.

No matter which method a test case is used to complete the UI transition, the tested app will eventually reach a stable interface unless the system collapses with FC (Force Close). When the app settles down to a steady-state, there are three possible situations: reach an unvisited Activity, reach an unvisited Fragment, reach a visited interface. These three situations may trigger the update of AFTM and/or the UI queue through a kind of evolutionary manner. The ways of handling different situations are described as below:

Case 1 – Reaching an unvisited Activity. When FragDroid reaches a new Activity interface, a new item will be added to the UI queue. The operation list of this item contains the operation list of its previous item and the operation information converting from the previous item to here. Also, if the function `getFragmentManager()` or `getSupportFragmentManager()` is found in the current Activity, it means the existence of dynamic switching between this Activity and some Fragment. Therefore, based on the amount of the Fragments having dependency with this Activity, the same amount of items will be added to the UI queue. The operation lists of these items contain the operation information reaching this Activity and the approach of switching from this Activity to this Fragment. In Case 1, we use the Java reflection mechanism as the switching method by default.

Case 2 – Reaching an unvisited Fragment. There are two ways for FragDroid to reach a new Fragment: clicking event

and Java reflection mechanism. The Java reflection mechanism offers the functionality of obtaining all inside information of a known class, such as fields and methods, and it allows to invoke any field or method of a given object. With the aid of reflecting the `FragmentManager` class in the target Activity, the corresponding `FragmentTransaction` class could be constructed. The sub-classes of Fragment involved in the current Activity will be instantiated on Java Virtual Machine by reflection, and then they are filled into the corresponding switch functions of `FragmentTransaction` class. Finally the transition between Fragments could be implemented by executing the method `commit` in the `FragmentTransaction` class. Besides, if a new Fragment could be visited through a clicking event, this explicit clicking process will take the place of the implicit reflection mechanism and is set as the initial operation for transition during element generations.

Case 3 – Reaching a visited interface. If the testing app reaches a visited interface, FragDroid will complete the input fields and get all coordinates of the controls that can be clicked on this interface. Then, clicking events will be injected by FragDroid from top to bottom, from left to right automatically. A clicking event may lead to the following situations: if the interface doesn't change, the clicking operation will move on; if a dialog box or a menu pops up, it will be removed by clicking on blank space and continues the clicking operation; if the interface changes, the new interface will be terminated by killing current execution and the testing app will be restarted and execute under the click operation until all clicking events are completed; if the app crashes, the testing app will be restarted to the current interface and execute under clicking operations.

B. Queue Generation & Test Case Generation

For the whole evolutionary test, the UI transition queue is maintained on the basis of AFTM in a width-first strategy. Each dynamically generated item in the queue is the information on the transition from one interface to another. This procedure can be divided into two core modules – queue update and test case generation.

In the beginning, FragDroid uses the original AFTM to initialize the UI transition queue. In the UI transition queue, the data structure of each UI queue item contains four properties: the way of reaching a certain interface (Activity or Fragment), start interface, target interface, and an operation list storing the concrete operations from the start interface to the target interface. Once AFTM is updated, the queue will also be updated.

The test case generation module transforms the items in the UI queue into executable test cases. The template of test case based on the library of Robotium is accomplished with the information inside the items. One of the most difficult parts is to generate the test case of mandatory switching to hidden or unvisited Fragments. In the process of translating the operation list into Java code for test cases, if no explicit operation can be used for interface transition, the Java reflection mechanism will be utilized. The concrete process of reflection is to reflect the Activity class that the target Fragment belongs to, then determine whether there exists `getSupportFragmentManager()` or `getFragmentManager()` in that class. Respectively, the Activity inherits from `android.app.Activity` or `android.support.v4.app.Activity`. The corresponding `FragmentManager` is obtained to execute `beginTransaction()` and reflect the Fragment classes to switch. Finally, the switching function is constructed and executed with the Fragment container's resource-ID.

C. Test Termination Condition

As shown in Figure 4, FragDroid generates items for the UI queue based on AFTM. After that, the items are further compiled to test cases and executed for dynamic testing. The state information is extracted from the testing results to update AFTM. The update of AFTM triggers the next round of execution. Once the UI transition queue is empty, and AFTM is no longer updated, it means all test cases have been tested, that is the end of the loop.

To achieve a higher coverage rate and avoid omission, if there are Activities that haven't been visited (according to the final AFTM), FragDroid will forcibly invoke them through empty Intents. If the invocation succeeds, these Activities will be added into AFTM with normal processing as the second loop phase. When this loop stops (empty UI queue and no update for AFTM), the whole test terminates.

VII. Experiment and Evaluation

We have implemented a full-featured prototype of FragDroid and carried out a series of experiments. Our experiments primarily concentrate on the effectiveness of our framework. Also, to better illustrate the practicability of our framework, we choose the sensitive API call analysis as a showcase.

A. Dataset

We downloaded and analyzed 217 popular apps (more than 500,000 downloads) from 27 categories of Google Play. The categories include Tools (21 apps), Entertainment (21 apps), News Magazine (16 apps), Business Office (15 apps), Books and Reference (14 apps), etc. The preliminary code analysis discovered 91% of them use Fragment components.

However, since some apps were encrypted or protected (with packer), they cannot be analyzed and have to be ruled out in the dependency extraction phase. Also, some apps failed in the dynamic testing due to the issues of permissions. Nevertheless, these problems are out of the scope of our framework, so not all apps were considered in the experiment. Finally, we selected 15 apps from these 217 apps for further analysis.

B. Coverage

In total, FragDroid successfully covered 66% Fragments and 71.94% Activities. Table I shows the coverage rate of FragDroid in aspects of Activities, Fragments, and the Fragments in visited Activities. In this table, each data column consists

TABLE I: Coverage of Activities and Fragments Detection

Package Name*	Downloads	Activities			Fragments			Fragments in Visited Activities		
		Visited	Sum	Rate	Visited	Sum	Rate	Visited	Sum	Rate
au.com.digitalstampede.formula	50,000+	1	2	50.00%	2	2	100.00%	1	1	100.00%
com.adobe.reader	100,000,000+	7	13	53.85%	5	5	100.00%	2	2	100.00%
com.advancedprocessmanager	10,000,000+	5	7	71.43%	10	10	100.00%	10	10	100.00%
com.aircrunch.shopalerts	1,000,000+	7	10	70.00%	8	13	61.54%	4	6	66.67%
com.c51	5,000,000+	28	35	80.00%	2	3	66.67%	2	3	66.67%
com.cnn.mobile.android.phone	10,000,000+	16	23	69.57%	3	10	30.00%	2	4	50.00%
com.happy2.bbmanga	1,000,000+	2	5	40.00%	3	5	60.00%	0	2	0.00%
com.inditex.zara	10,000,000+	7	9	77.78%	7	15	46.67%	2	10	20.00%
com.mobilemotion.dubsmash	100,000,000+	10	11	90.91%	0	3	0.00%	0	3	0.00%
com.ovuline.pregnancy	1,000,000+	17	27	62.96%	8	37	21.62%	8	26	30.77%
com.weather.Weather	50,000,000+	13	17	76.47%	1	1	100.00%	1	1	100.00%
com.where2get.android.app	500,000+	9	16	56.25%	4	8	50.00%	0	4	0.00%
imoblife.toolbox.full	10,000,000+	14	14	100.00%	8	9	88.89%	4	5	80.00%
net.aviascanner.aviascanner	1,000,000+	7	7	100.00%	4	4	100.00%	4	4	100.00%
org.rbc.odb	1,000,000+	4	5	80.00%	5	8	62.50%	2	3	66.67%

* All APKs are available on Google Play Store

of three sub-columns: Visited, Sum, and Rate. The data of the "Visited" column records the number of the corresponding elements (e.g., Activities) successfully tested by FragDroid on each app; the column "Sum" summarizes the number of such elements discovered in the phase of Static Information Extraction, and the column "Rate" lists the ratio of visited elements in all elements (i.e., the coverage rate) on a certain level. Taking the category of Fragments in Visited Activities as an example, it shows the result of testing apps via FragDroid on the level of Fragments in Activities that are visited by FragDroid, including the number of visited Fragments in tested Activities, the number of all Fragments in tested Activities and the percentage of visited Fragments in all Fragments in tested Activities.

1) Activity Coverage Analysis: Activities are the primary containers for UI layouts, and Fragments cannot live without Activities. Although our analysis focus is not the level of Activities but the level of Fragments, exploring more Activities could lead to a higher coverage rate for Fragments. The list of Activities of an app is extracted from its Manifest file during the static analysis, and the isolated ones (which cannot be switched to/from other interfaces) are excluded. FragDroid gives the number of visited Activities and logs all visited Activities.

However, there are some situations that decrease coverage rate of FragDroid for Activities.

- Current developers usually use the material design mode to develop apps, so that the transition of Activities in navigation view drawer cannot be operated directly [18], such as *com.cnn.mobile.android.phone* and *com.aircrunch.shopalerts*. To solve this problem, we have applied the mandatory starting in the phase of evolutionary test case generation. However, since this operation does not take the context and Intent into account, some Activities are still not detected by FragDroid.
- Some Activities require strictly accurate input to move to the next step. Some special inputs like address names are not given manually in advance, as a result, some apps (such as *com.weather.Weather*) cannot be tested smoothly.
- There are apps like *com.aircrunch.shopalerts*, *com.where2get.android.app*, and *com.inditex.zara*, which have action bars [19], and numerous pop operations are triggered in the app bars, such as *com.adobe.reader*. These operations usually lead to the changes of normal interfaces and interrupt normal test case generation.

2) Fragment Coverage Analysis: The sum of Fragments is found by FragDroid is based on the method of getting effective Fragments introduced previously. Fragments rely on Activities. A few Activities are inaccessible in some apps, and there are some related research on the discovery of Activities, as summarized in Section IX. Since this paper focuses on the probability of exploring Fragments, our framework FragDroid implements calculating and logging all the Fragments of visited Activities.

However, there are several Fragments instantiated or loaded directly without using `FragmentManager`. In this scenario, FragDroid cannot determine whether the Fragment is a real loading, as the failing case *com.mobilemotion.dubsmash*. Additionally, another app *com.inditex.zara* failed due to the missing parameters transmitted in the reflection mechanism.

3) Fragment in Visited Activity Coverage Analysis: As discussed above, some of Activities are not accessible for testing, which means the Fragments involved in such Activities are also inaccessible. In the experiment, the average coverage rate of FragDroid for Fragments in visited Activities is more than 50%, and for a third of tested apps, this coverage rate has reached 100%. It confirms that the basic test unit of FragDroid has been specified from Activity to Fragment. Furthermore, this framework takes the additional logic caused by the importing of Fragments into consideration, as a result, it is of good compatibility with Fragments.

C. Sensitive API Invocation Analysis

The user privacy disclosure in Android is a common concern, caused by the abuse of permissions and the transfer of sensitive APIs. There are some solutions to solve this

problem such as native permission management and dynamic permission reminder in Android 6.0 to remind the users, which limits the abuse of authority. Nevertheless, most sensitive operations are allowed by default at the time of installing an app. As an Android testing framework, FragDroid offers a way to detect sensitive operations. We select some common sensitive operation functions defined by XPrivacy [20] for testing. The sensitive APIs concerned in this experiment are mainly related to the information or operations like account, identification, Internet, IPC, location, media, network, phone, store. In Table II, we could find the discovered invocation relations between sensitive APIs and Activities and/or Fragments on 15 tested applications. For an individual tested app, each sensitive API listed has three possible situations, as shown below, where the symbols represent that a sensitive operation or permission is invoked by Activity and/or Fragment.

- **Invoked by Activity:** ◖
- **Invoked by Fragment:** ◗
- **Invoked by Both Activity and Fragment:** ⊙

The results show 46 sensitive APIs were found by FragDroid. Also, the API invocations associated with Fragments account for 49% of the total invocations. The traditional approaches based on Activity have to miss at least 9.6% of API calls invoked in Fragments.

VIII. Limitation and Future Work

At first, the small sample size in the experiment for implementing FragDroid limits the evaluation of analysis results. Secondly, there are some specific development methods and techniques which are not counted in FragDroid. Hence, a small portion of Activities and Fragments are missed out during the test. Moreover, FragDroid proposed in this paper is an Android application testing framework focusing on Fragments, without regard to other factors affecting its performance, such as input generation in test cases.

In the future, to optimize the effectiveness of FragDroid, more techniques related to app developing and testing will be considered, and better input generation methods will be integrated into it.

IX. Related Work

Recent works on testing tools have taken numerous, diverse approaches to achieve different results. Due to that Android apps often suffer from cross-platform and cross-version incompatibilities, it is costly for the manual app analysis. Moreover, the rapid growth of the number of apps makes those encountered different security threats and malicious attacks. So, there has been a great deal of research in static analysis and test of Android apps. We can find many tools do test input generation and extract the model of the app automatically. There are also some systems leveraging automated UI interfaces to trigger the functions [21], [22], [23], especially getting useful information or analyze data of apps by targeting specific Activities.

SmartDroid [24], applying both static and dynamic analysis, is a tool to discover and test UI trigger conditions using a hybrid approach. During static analysis, it creates an Activity switch path that leads to the sensitive API calls. In dynamic analysis, SmartDroid traverses the view tree of an Activity and triggers the event listeners while waiting for each UI element to arise. Very similar to our work, SmartDroid determines whether a new Activity is on the switch path when the event listener invokes the start of it. If not, it will block the call to that Activity and continue to traverse the current Activities view tree until the correct element is activated. It will exclude the calls to dynamically loaded code or native libraries because of only relying on static analysis to reveal sensitive behaviors. In addition, SmartDroid requires modification to the SDK as well as the modified emulator.

AndroidRipper [25] is an automated test technique which uses their Graphical User Interface (GUI) for Android apps. AndroidRipper is based on a user-interface driven ripper that aims at automatically exploring the GUI of apps in a structured manner. It is evaluated on open-source Android apps. The results show that those test cases, which are based on GUI, are able to detect severe and previously unknown faults in the underlying code, and its structured exploration outperforms a random approach.

Dynodroid [15] proposes a system to interact with UI widgets dynamically. The authors implemented a mechanism that attempts to generate a sequence of intelligent UI interactions and system events through observing the UI layout, composing and selecting a set of interactions, and executing those actions. Dynodroid leverages the Hierarchy Viewer, a tool packaged with the Android platform to infer a UI model during execution, to determine an Activity layout. We note that if the user intends to enable this capability, it is essential to make changes to the SDK source code. Finally, but most importantly, Dynodroid requires the tester has access to the source code of an app, as the use of the Android instrumentation framework is necessary. Contrarily, CuriousDroid can test any APK file without source code on account of the fact that it instruments the app bytecode dynamically.

A3E [26] proposes a system for UI exploration of Android apps that has two approaches. One is Targeted Exploration which generates a CFG during static analysis and then uses the CFG to develop a strategy for exploration by targeting specific Activities. Another is depth-first Exploration which attempts to mimic user interactions to drive execution in a more systematic, albeit slower, way. We can see that A3E is not proper for large-scale test on account of the fact that its long test time required for each app. A3E was tested on only 25 apps, and during each test, it costs an average runtime of 87 minutes for targeted exploration method and 104 minutes per app averagely for the depth-first exploration.

AspectDroid [27] is an app-level system designed to investigate Android apps for possible unwanted Activities. AspectDroid is comprised of application instrumentation, automated test, and containment systems. By using static bytecode instrumentation, AspectDroid weaves monitoring code into an existing app and provides data flow and sensitive API usage as well as dynamic instrumentation capabilities. The newly repackaged app is then executed either manually or via

an automated test module. Finally, the flexible containment provided by AspectDroid adds a layer of protection so that malicious Activities can be prevented from affecting other devices. The accuracy score of AspectDroid, when tested on 105 DroidBench corpus, shows it can detect tagged data with 95.29%.

TrimDroid [13] is a framework to generate test cases for GUI test of Android apps, with the ability to achieve a comparable coverage as it possible under exhaustive GUI test using fewer test cases. The work of TrimDriod uses prime path coverage to generate the event sequences and generate the inputs for GUI widgets in a combinatorial fashion rather than using randomly generated input. Similar to FragDroid, TrimDroid extracts the interface models and Activity transition models of apps. Note that TrimDroid does not involve Fragments, but FragDroid considers them.

Those testing tools mainly research the Activity layer. However, based on a significant amount of analysis we have done, there are some sensitive operations and embedded WebView as well as security threats brought by library reference in Fragments. Compared with those tools, FragDroid completes a lot of work on Fragments and provides a model, including not only Activities but also Fragments and all the transition relationship, to be utilized to understand the architecture of the entire app better and do some security research. The test result shows FragDroid can be considered as a more comprehensive and brand-new tool.

X. Conclusion

In this paper, we propose FrogDroid, the first Android automated UI testing framework supporting both Activity and Fragment analysis. In this framework, an Activity & Fragment Transition Model is introduced, with the use of which, test cases can be generated through automated UI interaction. FragDroid runs test cases to visit Activities and Fragments in the tested app for detecting security information, such as sensitive APIs and potential vulnerabilities. The analysis of top-rank apps reveals a high proportion of apps applying Fragments. During the experiment, FragDroid is applied to 15 selected apps involving Fragments. On average, the coverage rate of FragDroid on Activity is 71.94% while that on Fragment reaches 66%. There are 269 invocations of sensitive APIs detected from 15 tested apps, and nearly half of them have an association with Fragment.

XI. Acknowledge

We thank our shepherd Ilir Gashi for his guidance on improving this paper and anonymous reviewers for their insightful comments. This work is partially supported by National Natural Science Foundation of China (91546203), the Key Science Technology Project of Shandong Province (2015GGX101046), the Shandong Provincial Natural Science Foundation (ZR2014FM020), Major Scientific and Technological Innovation Projects of Shandong Province, China (No.2017CXGC0704) and Fundamental Research Fund of Shandong Academy of Sciences (NO.2018:12-16).

References

[1] "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html.
[2] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81.
[3] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 347–356.
[4] "Espresso," https://developer.android.com/training/testing/espresso/index.html.
[5] "Appium," http://appium.io/.
[6] "Robotium," http://www.methodsandtools.com/tools/robotium.php.
[7] "Introduction to activities," https://developer.android.com/guide/components/activities/intro-activities.html.
[8] D. Hackborn, "The android 3.0 fragments api," https://android-developers.googleblog.com/2011/02/android-30-fragments-api.html, 2011.
[9] "Fragment," https://developer.android.com/guide/components/fragments.html.
[10] "Fragmenttransaction," https://developer.android.com/reference/android/app/FragmentTransaction.html.
[11] "Apktool," https://ibotpeaches.github.io/Apktool/.
[12] "Jd-core-java," https://github.com/nviennot/jd-core-java.
[13] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 559–570.
[14] J. Chen, X. Cui, Z. Zhao, J. Liang, and S. Guo, "Toward discovering and exploiting private server-side web apis," in *Web Services (ICWS), 2016 IEEE International Conference on*. IEEE, 2016, pp. 420–427.
[15] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 224–234.
[16] "Android debug bridge," https://developer.android.com/studio/command-line/adb.html.
[17] "Apache Ant," https://ant.apache.org/.
[18] "Navigation drawer," https://developer.android.com/training/implementing-navigation/nav-drawer.html.
[19] "App bar," https://developer.android.com/training/appbar/index.html.
[20] "Xprivacy," https://github.com/M66B/XPrivacy/blob/master/res/values/functions.xml.
[21] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.
[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258–261.
[23] 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.
[24] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12, 2012, pp. 93–104.
[25] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012, pp. 258–261.
[26] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *SIGPLAN Not.*, vol. 48, no. 10, pp. 641–660, Oct. 2013.
[27] A. Ali-Gombe, I. Ahmed, G. G. Richard, III, and V. Roussev, "Aspectdroid: Android app analysis system," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16, 2016, pp. 145–147.

TABLE II: Sensitive Operations Detection

Sensitive APIs \ Package Name / Usage		com.adobe.reader	com.aircrunch.shopalerts	org.rbc.odb	au.com.digitalstampede.formula	com.happy2.bbmanga	com.c51	com.weather.Weather	com.advancedprocessmanager	com.inditex.zara	com.cnn.mobile.android.phone	com.mobilemotion.dubsmash	com.ovuline.pregnancy1	com.where2get.android.app	imoblife.toolbox.full	net.aviascanner.aviascanner
Browser	browser/Downloads	D							D							
Identification	identification//proc		D		⊙		◖	◖	⊙		◖		⊙	◖	◖	◖
	identification/getString		⊙		⊙		⊙	◖					⊙	◖	⊙	◖
	identification/SERIAL		D	◖	⊙	◖	◖	◖	◖	⊙	◖	◖	⊙	◖	◖	◖
Internet	internet/connect	◖			⊙	⊙	⊙	⊙			◖		⊙			
	internet/Connectivity.getActiveNetworkInfo	◖	⊙		⊙	⊙	⊙	◖	D	D	◖		⊙		◖	⊙
	internet/Connectivity.getNetworkInfo							◖	D	◖	◖		⊙		◖	
	internet/inet					◖	◖	◖	◖							
	internet/InetAddress.getAllByName		⊙		⊙	⊙	⊙	⊙		D	◖		⊙		◖	⊙
	internet/InetAddress.getByAddress							◖	D	⊙	◖		⊙			
	internet/InetAddress.getByName	◖	⊙	◖	⊙		⊙	◖		⊙	◖	◖	⊙	◖	◖	◖
	internet/IpPrefix.getAddress								D							
	internet/LinkProperties.getLinkAddresses		⊙		⊙	⊙	⊙	◖	⊙	⊙	◖		⊙		⊙	⊙
	internet/NetworkInfo.getDetailedState							◖			◖				◖	
	internet/NetworkInfo.isConnected	D	◖			◖		◖	D				⊙		⊙	
	internet/NetworkInfo.isConnectedOrConnecting	◖	⊙		◖	D	⊙	◖	D	⊙	◖		⊙		◖	⊙
	internet/NetworkInterface.getNetworkInterfaces				⊙								⊙		◖	◖
	internet/WiFi.getConnectionInfo							◖		D			⊙			
IPC	ipc/Binder		D	◖	⊙	◖	◖	◖	◖	◖	◖	◖	⊙	◖	⊙	◖
Location	location/getAllProviders						◖	◖					⊙			
	location/getProviders							◖		⊙						
	location/isProviderEnabled		D				◖	⊙								◖
	location/requestLocationUpdates				⊙		◖	◖		⊙			⊙			
Media	media/Camera.setPreviewTexture									D						
	media/Camera.startPreview						◖			D						
Messages	messages/MmsProvider								D							
Network	network/NetworkInterface.getInetAddresses												⊙		◖	◖
	network/WiFi.getConfiguredNetworks														⊙	
	network/WiFi.getConnectionInfo							◖					⊙			
Phone	phone/Configuration.MCC		⊙	◖	⊙	◖	◖	⊙	⊙	◖	◖		⊙	◖	⊙	⊙
	phone/Configuration.MNC		⊙	◖	⊙	◖	◖	⊙	⊙	◖	◖		⊙	◖	⊙	⊙
	phone/getDeviceId				⊙		◖	◖		⊙					◖	
	phone/getNetworkCountryIso	◖	D				◖	◖			◖		⊙		◖	
	phone/getNetworkOperatorName				⊙			◖		D	◖		⊙	◖	◖	
Shell	shell/loadLibrary		D		⊙		◖	◖		⊙	◖			◖	◖	
Storage	storage/getExternalStorageState	◖	D			⊙		◖		⊙					◖	
	storage/open									⊙	◖		⊙		⊙	
	storage/sdcard		⊙		⊙			◖	◖	⊙	◖	◖	⊙	◖	◖	
System	system/getInstalledApplications								D							
	system/getRunningAppProcesses									D	◖				◖	
	system/queryIntentActivities	◖			⊙											
	system/queryIntentServices							◖			◖		⊙			
View	view/getUserAgentString	◖			⊙											
	view/initUserAgentString	◖	⊙		⊙		◖	◖			◖				◖	
	view/loadUrl		⊙					◖			◖				◖	
	view/setUserAgentString		⊙					◖			◖				◖	

* ◖ Activity D Fragment ⊙ Activity and Fragment