Data Flow Analysis on Android Platform with Fragment Lifecycle Modeling

Yongfeng Li[1(✉)], Jinbin Ouyang[1], Shanqing Guo[2], and Bing Mao[1]

[1] State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China
jsliyongfeng@gmail.com, oyjb1992@gmail.com, maobing@nju.edu.cn

[2] School of Computer Science and Technology, Shandong University, Jinan, China
guoshanqing@sdu.edu.cn

Abstract. Smartphones carry a large quantity of sensitive information to satisfy people's various requirements, but the way of using information is important to keep the security of users' privacy. There are two kinds of misuses of sensitive information for apps. On the one hand, careless programmers may leak the data by accident. On the other hand, the attackers develop malware to collect sensitive data intentionally. Many researchers apply data flow analysis to detect data leakages of an app. However, data flow analysis on Android platform is quite different from the programs on desktop. Many researchers have solved some problems of data flow analysis on Android platform, like Activity lifecycle, callback methods, inter-component communication. We find that Fragment's lifecycle also has an effect on the data flow analysis of Android apps. Some data will be leaked if we don't take Fragment's lifecycle into consideration when performing data flow analysis in Android apps. So in this paper, we propose an approach to model Fragment's lifecycle and its relationship with Activity's lifecycle, then introduce a tool called FragDroid based on FlowDroid [7]. We conduct some experiments to evaluate the effectiveness of our tool and the results show that there are 8% of apps in our data set using Fragment. In particular, for popular apps, the result is 50.8%. We also evaluate the performance of using FragDroid to analyze Android apps, the result shows the average overhead is 17%.

Keywords: Data flow · Fragment · Android · Program analysis

1 Introduction

With the progress of technology, smartphones have pervaded into all aspects of human life, and have become an indispensable part of daily life. Compared with the traditional PC devices, smartphones carry more user privacy data, such as location information, contact information, fingerprint information, text message records, which brings endless attacks against smartphones. The security protection of smartphones has become a problem which needs to be solved urgently.

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 637–654, 2017.
DOI: 10.1007/978-3-319-59608-2_35

According to the recent report [1], in the current smartphone markets, Android platform market share has been far more than the iOS platform. This means protecting the privacy of Android users is very important.

On Android platform, there is a variety of malware. In the research of Jiang's team [2], they classify the malware based on behavior. In their malware classification, there are many kinds of malware which collect users' privacy information and leak it out. Sometimes, privacy information is not leaked by malware intentionally. Developers always use some third-party libraries to develop an Android app conveniently, which is hard for developers to know the details of data flow in the libraries. And when they pass privacy information to the library procedure, information may be leaked. The library itself can also lead to information leakage.

To protect the privacy information, there is a kind of technology called taint analysis, whose main task is to record the data flow relationship among some specific objects. In taint analysis, before propagating the data flow, some nodes called sources (in data leakage, these are sensitive APIs which get information like GPS, location, etc.) should be specified. During the data flow propagation, taint analysis will check if the data flow reach nodes called sinks (APIs which send messages). Through taint analysis, privacy information leakage which violate predefined rules will be found. There are two approaches to perform taint analysis on Android platform: dynamic analysis and static analysis. Some dynamic analysis techniques like TaintDroid [3], Droidscope [4] have been proposed. These approaches all are suffering from the code coverage problem, that is, when running an app, some code may not be executed. Moreover, as mentioned in [5], malware can use the runtime information to decide whether it is running on a monitor or not. Then, it can decide whether to trigger malicious behaviors or not.

Static analysis is performed with scanning the apps instead of executing programs, which avoids the problems mentioned above. However, it demands to emulate the runtime state of an app approximately. Previous researchers [6–11] have proposed some approaches to solve the problems in static analysis on Android platform. Chex [6] is a static analysis system designed to solve the component hijacking problems of Android apps. To handle the multiple entry points, Chex conducts data flow analysis for code reachable from each entry point, and then combines these results to find data flow between code splits. FlowDroid [7,8] models the taint-analysis problem within the IFDS [12] framework for inter-procedural distributive subset problems. FlowDroid generates a dummy main method for each app to model the control flow transfers between component lifecycle methods. FlowDroid also models the control flow of callback methods in a dummy main method. Amandroid [9] and IccTA [10] handle the inter-component communication when performing data flow analysis. AmanDroid calculates all objects' points to information, while IccTA handles the situation when Activity is not the target of ICC based on FlowDroid. EdgeMiner [11] conducts a deep study of callback methods in Android system. EdgeMiner proposes an automatic approach to extract callback methods and their corresponding registration methods in Android system. FlowDroid can apply EdgeMiner's result to get more precise data flow information.

The fragment introduced in Android 3.0(API level 11) is mainly to support a more dynamic and flexible UI design for the large screen(such as tablet PC). Because of the much larger screen of tablets' compared with that of smartphones, more space can be used to combine and exchange UI components. On account of the Activity layout divided into fragments, you can modify its appearance and keep the changes in return stack which is managed by Activity itself. Fragments is part of the behavior or the user interface of Activity. We can use multiple fragments combination in an Activity to build multiple pane UI, and reuse a fragment in multiple Activities. We can put the fragment as a modular part of the Activity, which has its own lifecycle and can receive their own input events. Moreover, we can dynamically add, replace, and remove some fragments. None of the previous researchers have described Fragment's lifecycle has an effect on data flow analysis. When performing data flow analysis, some data flow will be missed without taking Fragment's lifecycle into consideration, which will lead to false negative when analyzing data leakage in apps. Moreover, malware can also adopt Fragment's lifecycle to evade the detection method based on data flow and control flow analysis. Moreover, Fragment's lifecycle is not independent as it depends on Activity's lifecycle. So we also model the interaction between Activity and Fragment. Malware is out of the scope of this paper, so we don't discuss it in this paper.

To summarize, this paper makes the following contributions:

– We find that Fragment's lifecycle has an effect on data flow analysis on Android apps. And we do some research to reveal the relationship between Fragment's lifecycle and Activity's lifecycle.
– We model all the Fragments' and Activities' lifecycle control flow transfers in a control flow graph, then we make an extension on Flowdroid [7]. All of the lifecycle methods are contained in a dummy main method. With using the extended tool, we can perform information leakage detection without false negative caused by Fragment's lifecycle.
– We make an in-depth evaluation of the extended tool. The experiments' result include the statistics of the Fragment usage in Android apps and the runtime performance after modeling the Fragment's lifecycle.

2 Background and Motivation

2.1 Background

In Android, an application's execution is driven by system events. When an event occurs, Android system invokes the predefined method which implemented by developers. Android adopts a component-based mechanism to simplify the development of apps. There are four kinds of application components: Activity, Service, Content Provider, Broadcast Receiver. Each app is composed of many application components, and the components' execution is controlled by system according to events. When performing data flow analysis in Android apps, components' lifecycle must be taken into consideration. Previous researchers [7–10] have modeled the lifecycle of four main application components. But in 3.0, Android introduces Fragment to support more dynamic and flexible UI designs on large screens, such as tablets. In Android apps, Fragment is always included in an Activity which has its own lifecycle, so does the Fragment. The lifecycle of Fragment and its relationship with Activity's lifecycle are described in Fig. 1.

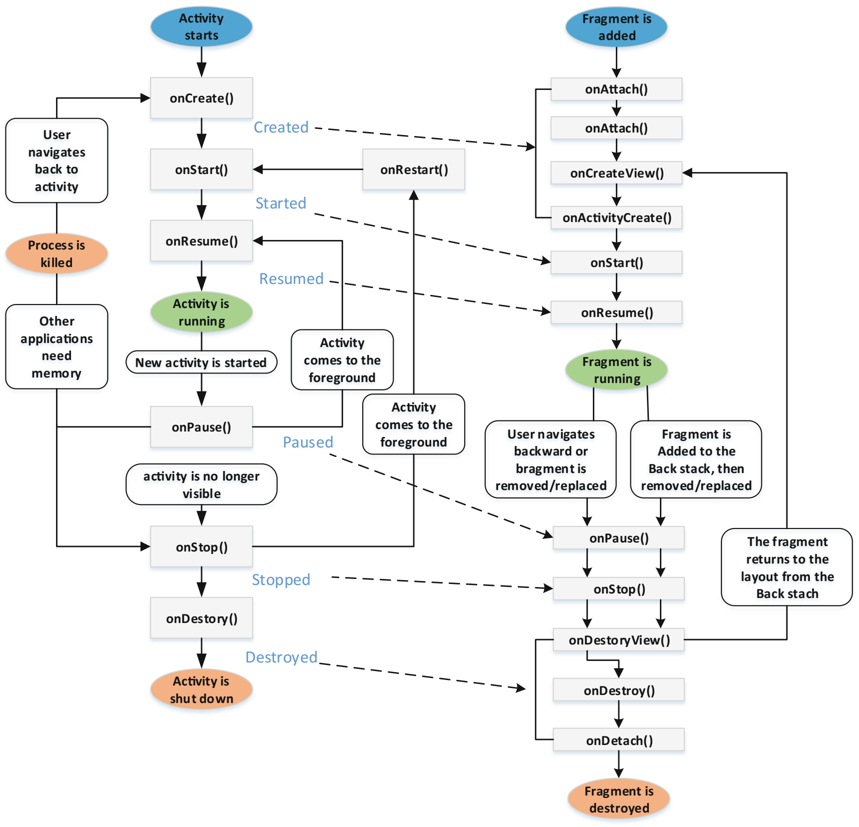


Fig. 1. Fragment lifecycle

As is depicted in Fig. 1, when a Fragment starts, onAttach(), onCreate(), onCreateView(), onActivityCreated(), onStart() and onResume() will be invoked by Android system one by one. When the app is paused, for example, Fragment's onPause() method will be invoked if the user presses the home button. When the memory is low, Android system will recycle some memory, so the onStop() method will be invoked. When the user navigates back to the app, Fragment's onStart() and onResume() method will be invoked to restore the Fragment. When the user kills the app, onDestroyView(), onDestroy(), onDetach() will be invoked. Moreover, the Fragment's lifecycle depends on Activity's lifecycle.

Thus we can't use Fragment alone. Activity's lifecycle dominates Fragment's lifecycle. So the Activity starts and pauses before Fragment, while stops and destroys after Fragment. Thus, Activity's onCreate(), onStart(), onResume() will be invoked before Fragment's onCreate(), onStart() and onResume(). And Activity's onPause(), onStop(), onDestroy() will be invoked after Fragment's onPause(), onStop(), onDestroy().

2.2 Motivation

In this section, we demonstrate our motivation by introducing some code snippets. As is depicted in Fig. 2, there is an Activity named "LifecycleActivity" and a Fragment named "LifecycleFragment". "LifecycleActivity" overrides two lifecycle methods onCreate() and onPause(), "NativePhoneNumber" is a field in this class. "LifecycleFragment" overrides three lifecycle methods onAttach(), onResume() and onPause(). "LifecycleFragment" also has two fields named "NativePhoneNumber" and "attachedActivity". The first one stores string value, while the second one stores the reference of Activity this Fragment attached to. In "LifecycleActivity", onCreate() method invokes replace() method to attach a "LifecycleFragment" to this Activity. onPause() method invokes getLine1Number() and stores the phone number to field "NativePhoneNumber". In "LifecycleFragment", onAttach() method stores attached Activity's reference to field "attachedActivity". onResume() method passes the the field "NativePhoneNumber" of "attachedActivity" to its field "NativePhoneNumber". onPause() method invokes sendTextMessage() and sends out the value stored in "NativePhoneNumber".

FlowDroid models the application components' lifecycle in a dummy main method, but the Fragment's lifecycle is not in this dummy main method. So if we use FlowDroid [7] to detect data leakage in this app, it will report nothing. FlowDroid has generated control flow graph before data flow analysis and the Fragment's lifecycle methods are not in this graph, so data flow will not propagate out of these lifecycle methods. But actually, this app leaks the phone number through sending text message. AmanDroid [9] also can't detect this data leakage, because it doesn't consider Fragment's lifecycle as well.

From Fig. 1, we know that when "LifecycleActivity" starts, the system invokes onCreate() method, then the "LifecycleFragment" is attached to this Activity. At the same time, Android system invokes the lifecycle method onAttach() of "LifecycleFragment". In this method, the reference of "LifecycleActivity" is passed to "attachedActivity". When "LifecycleActivity" is activated, the onResume() method of "LifecycleFragment" is invoked, so the string value in "NativePhoneNumber" of "LifecycleFragment" is "default". At this moment, if user leaves "LifecycleActivity" Activity, the lifecycle method onPause() of "LifecycleFragment" will be invoked. It sends the value of "NativePhoneNumber" which is "default". It means the information leakage has not happened so far. Then the lifecycle method onPause() of "LifecycleActivity" will be invoked, so the string value in "NativePhoneNumber" of "LifecycleActivity" will be the phone number. When user navigates back to "LifecycleActivity", lifecycle

```
public class LifecycleActivity extends Activity
{
     public String NativePhoneNumber = "default";
     public void onCreate(Bundle savedInstanceState)
     {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);
          LifecycleFragment fragment = new LifecycleFragment();
          getFragmentManager().beginTransaction()
                         .replace(R.id.container, fragment).commit();
     }
     protected void onPause() {
          super.onPause();
          NativePhoneNumber = telephonyManager.getLine1Number();
     }

}

public class LifecycleFragment extends Fragment
{
     private String NativePhoneNumber = "default";
     private LifecycleActivity attachedActivity;
     @Override
     public void onAttach(Activity activity)
     {
          super.onAttach(activity);
          attachedActivity = (FragmentLifecycle) activity;
     }
     public void onResume()
     {
          super.onResume();
          NativePhoneNumber = attachedActivity.NativePhoneNumber;
     }
     public void onPause()
     {
          super.onPause();
          SmsManager sms = SmsManager.getDefault();
          sms.sendTextMessage("134444", null, NativePhoneNumber, null, null);
     }
}
```

Fig. 2. Motivation example

method onResume() of "LifecycleFragment" will be invoked, and the value of "NativePhoneNumber" in "LifecycleFragment" will be the phone number. If the user leave "LifecycleActivity" again, the lifecycle method onPause() of "LifecycleFragment" will be invoked, and the phone number will be leaked through text message this time.

From the description above, we can learn that this app gets phone number through the lifecycle method onPause() of "LifecycleActivity", and sends a text message with the phone number in the lifecycle method onPause() of "LifecycleFragment". During this process, the Activity's state changes many times. Some data flows will be lost if we don't model the control flow transfers between lifecycle methods when state changes, which will make malware evade detection with some state-of-art static analysis tools like FlowDroid and Amandroid. Besides control flow, some data dependencies between parameters of lifecycle methods also need to be handled carefully. For example, in lifecycle method onAttach() of "LifecycleFragment", its parameter, which is passed by Android system, is the Activity it attached to. When performing data flow analysis, we should take this data dependence into consideration.

2.3 Goals and Assumption

In this paper, we focus on Fragment's lifecycle and its effects on data flow analysis. We propose an approach to model Fragment's lifecycle and implement a tool named FragDroid. FragDroid is based on FlowDroid [7], so its data flow analysis has the same limits as FlowDroid. It can't deal with native code and decide the target objects or methods for java reflection.

3 System Design

We demonstrate FragDroid's work flow in Fig. 3. As is depicted in this figure, FragDroid takes six steps to analyze an app. First of all, it parses the manifest file and then the app's entry points like Activity, Service will be obtained. Then, FragDroid scans the Activity's lifecycle methods to find Fragment registrations. At the same time, FragDroid gets the Activity's layout xml file. Next, FragDroid parses the layout file to find the Fragment registration because Fragment can be attached in layout file as well. And then, Fragment gets some Fragments, but Fragment can also be declared in callback methods. So in the callback methods, some Fragments can be attached to Activity dynamically. The work flow will go back to step two unless no new fragments and callback methods can be found. At last, FragDroid generates a dummy main method to model the control flow transfer between the lifecycle methods of Fragment and Activity. A demo of dummy main method's control flow is shown in Fig. 4. After the dummy main method has generated, FragDroid builds the call graph and perform taint analysis just as FlowDroid does.

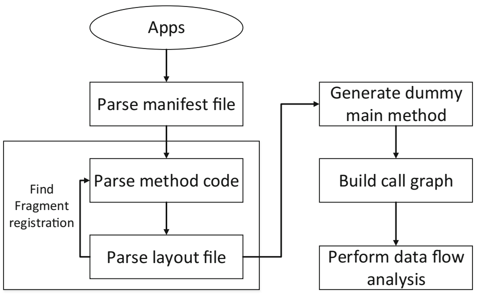

Apps
Parse manifest file
Find Fragment registration
Parse method code
Parse layout file
Generate dummy main method
Build call graph
Perform data flow analysis

Fig. 3. System architecture

4 Implementation

4.1 Identify Fragments Which Attached to Activity

In order to model the Fragment's lifecycle, we must find all Fragments what an app's Activities contain at first. In an Android application, Fragments can be

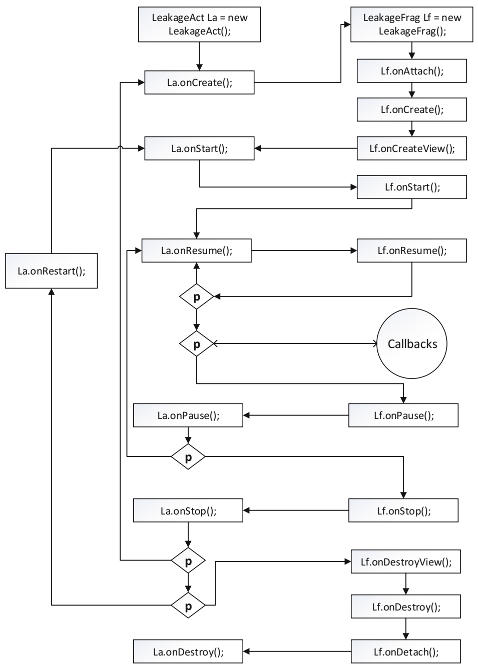


Fig. 4. Dummy main method control flow

attached to an Activity through two approaches. Firstly, developers can attach a Fragment to an Activity by declaring in the Activity's layout file. This file is an xml file, in which Fragment is declared by 'fragment' tag. For this kind of registration, we can scan the Activity's onCreate() method and get its layout file, then the fragment can be identified. Secondly, developers can attach Fragment to an Activity through some registration methods like FragmentTransaction.add() or FragmentTransaction.replace(). It is more complex here to find this kind of Fragment registration than in the first approach because these registration methods can be invoked during Activity's lifecycle. Moreover, users can change the Activity's user interface dynamically through callback methods like onClickListener(). Fragment registration can also happen in these callback methods. In order to find this kind of registration, we scan all lifecycle methods of the Activity implements. Then, we scan the Fragments' lifecycle methods and callback methods until no new Fragments' lifecycle and callback methods can be found.

4.2 Deal with Data Flow Between Activity and Fragment

In an Android app, Activity and Fragment have not only control flow relationships, but also some data flow dependencies. For example, as is shown in Fig. 2, the parameter of lifecycle method onAttach() in LifecyceFragment is passed by Android system. We need to handle this situation, otherwise some data flows will be lost. To solve this problem, we can modify the code of Fragment because we just need to maintain the data flow relationship instead of running the app. We create a new private field ‘attachedActivity’ in the class for Fragment’s onAttach() and getActivity() method. When generating dummy main method, for each Fragment, we pass the related Activity to Fragment’s onAttach() method. And then, in onAttach() method, the passed Activity is stored in ‘attachedActivity’. To get the correct Activity, we rewrite the getActivity() method whose return value is ‘attachedActivity’.

4.3 Create Dummy Main Method

After getting all Fragments each Activity contains, we need to generate a dummy main method to model the control flow transfers between lifecycle methods of Fragment and Activity. Figure 4 shows us the dummy main method’s control flow when an Activity only contains one fragment. If an Activity contains multiple Fragments, the situation will be more complex. We will describe how to solve it in the next section. In order to create a dummy main method whose control flow is like Fig. 4, we use a conditional jump instruction to model the control flow transfer among lifecycle methods.

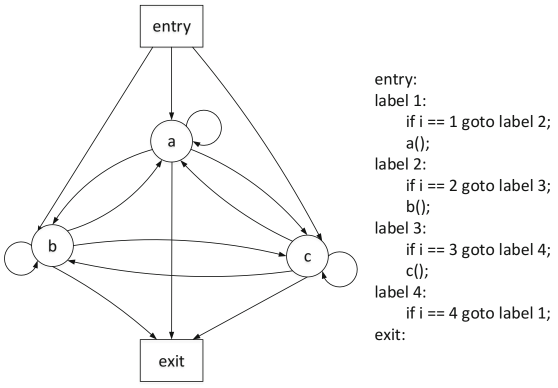
```
public static void dummyMainMethod()
{
    int $i0 = 0; LifecycleActivity$r1; LifecycleFragment $r2;
    label01:
      if $i0 == 1 goto label07;
      $r1.<LifecycleActivity: void onCreate()>();
    label02:
      if $i0 == 2 goto label03;
      $r2.<LifecycleFragment: void onAttach(Activity)>($r1);
    label03:
      if $i0 == 3 goto label04;
      $r2.<LifecycleFragment: void onResume()>();
    label04:
      if $i0 == 4 goto label05;
      $r2.<LifecycleFragment: void onPause()>();
    label05:
      $r1.<LifecycleActivity: void onPause()>();
      if $i0 == 5 goto label03;
    label07:
      return;
}
```

Fig. 5. Dummy main method IR code of motivation sample

We use the motivation example to demonstrate the creation of the dummy main method as is shown in Fig. 5. In this figure, at first, conditional value ‘i0’, Activity ‘r1’ and Fragment ‘r2’ is declared. In lable01, it creates a conditional jump whose target is label07 because this Activity may not be executed. If the condition is not met, ‘r1’ will invoke onCreate() method. In label02, the fragment’s onAttach() method is invoked depending on the conditional jump. Although we can get the Fragments which can be attached to an Activity, but we don't know which Fragments are attached to the Activity at an exact moment. We don't implement the Activity's onStart() and onResume() methods, so in label03, Fragment's onResume() is invoked. When the Activity is paused, the Fragment's onPause() method has been invoked before Activity's onPause() method. So in label04, Fragment's onPause() is invoked. And in label05, Activity's onPause() is invoked. The Activity's state can be resumed, so in label05 there is a conditional jump going to label03. In label07 the app is terminated.

4.4 Handle One Activity Carried with Multiple Fragments

In the last section, we have described how to create a dummy main method when an Activity only contains one Fragment. Actually, multiple Fragments can be attached to an Activity. Sometimes it is required to modify the entire page, but creating a new Activity is unnecessary. It can be efficient to use an Activity to manage multiple Fragments. News application, for example, can use a fragment to display article list on the left and another fragment to display the article on the right. Therefore, users do not need to use an Activity to select articles and use another Activity to read the article, but can choose articles within an Activity. In this section, we will show how to deal with this situation. If multiple Fragments are attached to an Activity, the lifecycle methods of Fragments are invoked according to the order of attaching these Fragments. Take an Activity with two fragments as an example, when the lifecycle methods of Fragments are invoked, as is shown in Fig. 4, the first attached Fragment's onAttach() to onActivityCreated() will be invoked after Activity's onCreate(). As there are two Fragments, the second attached Fragment's onAttach() to onActivityCreated() will be invoked after the first Fragment's. Fragments' onStart(), onResume(), onPause(), onStop() are also invoked in the Fragments' attached order. And the first Fragments' onDestroyView(), onDestroy(), onDetach() has been invoked before the second Fragment's. However, as is described in Sect. 4.1, Fragments in an Activity can be dynamically added or replaced by callback methods. Thus



```
entry:
label 1:
      if i == 1 goto label 2;
      a();
label 2:
      if i == 2 goto label 3;
      b();
label 3:
      if i == 3 goto label 4;
      c();
label 4:
      if i == 4 goto label 1;
exit:
```

Fig. 6. Control flow sequence of lifecycle methods when there are multiple fragments

we can't exactly know the order of how Fragments are added. In this paper, we assume the Fragments are attached in any order.

We show this kind of control flow transfer in Fig. 6. In this Figure, we assume there are three Fragments in an Activity. The vertexes labeled as a, b, c, have the same lifecycle method like onResume() of each Fragment. For the process from onAttach() to onActivityCreated(), we use an intermediate method to invoke them one by one. To make the dummy main method generate the control flow transfer as is shown in Fig. 6, we can generate the code shown in this figure. In the code, we use a conditional jump statement to model the execution. The lifecycle methods can be executed in any order by emulating different conditions.

5 Evaluation

5.1 Dataset and Experiment Setup

We collect 19342 apps from three popular Android markets (Baidu [17], Xiaomi [18] and Anzhi [19]). In order to measure the amount of Fragments in the most popular apps, we also select 887 apps from baidu market according to their downloads. To test the efficiency of FragDroid, we develop some test apps based on lifecycle methods of Fragment and Activity which override different lifecycle methods.

We conduct experiments on a computer equipped with Intel(R) Core(TM) i7-4770k CPU(3.5 GHz) and 16 GB of physical memory. The operation system is Windows 7.

5.2 Summary of Fragment Usage in App

The experiment results of Fragment usage of apps in the two data sets mentioned in the last section are shown in Figs. 7 and 8. In the first app data set, 1557 apps in 19342 cases use Fragment. This means, for an ordinary app, the probability of using Fragment is 8%. In the second app data set, 451 apps in 887 cases use Fragment, the probability is 50.8%. We also give the statistic result for multiple fragments can be attached to one activity in Fig. 9. According to these figures, we conclude that the more popular the app is, the higher its possibility of using Fragment is.

Figure 7 lists the distribution of the number of Fragments in the first app data set. From the figure, we find that 45.8% of apps which use Fragment only have one kind of Fragment, and Activities share the same user interface provided by this Fragment. Most of apps (91.9%) have less than 10 kinds of Fragments. Figure 8 shows the result of the second app data set, we can see that more kinds of Fragments are contained in one app, 28.3% apps have more than 10 kinds of Fragment classes.

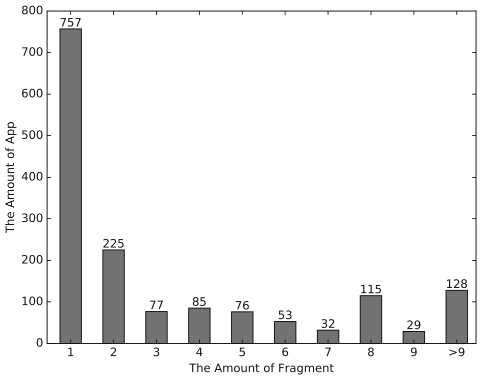


Fig. 7. Distribution of the number of fragments

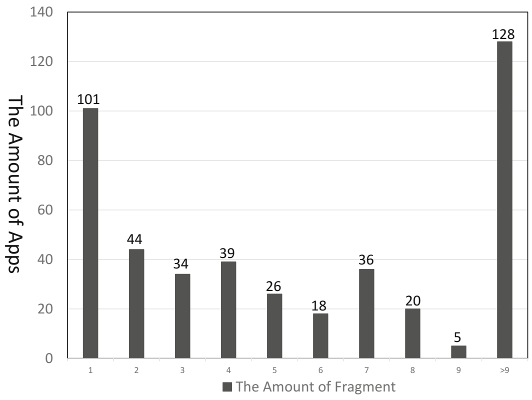
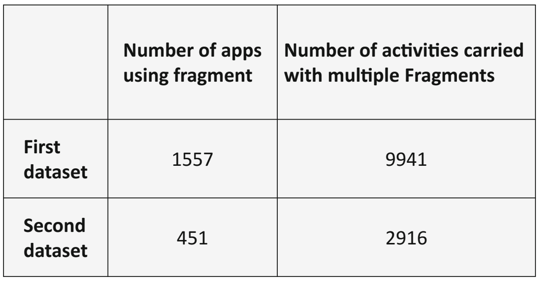


Fig. 8. Distribution of the number of fragments for popular apps



	Number of apps using fragment	Number of activities carried with multiple Fragments
First dataset	1557	9941
Second dataset	451	2916

Fig. 9. Distribution of the number of activities carried with multiple fragments

5.3 Data Leakage Results

The result of data leakage in the second app data set is shown in Fig. 10. This figure contains the results of analyzing and do not analyzing Fragment. In this figure, we find that when we don't consider Fragment lifecycle, 47.4% of apps report more than 150 source to-sink pairs. After Fragment's lifecycle is modelled, 57.3% of apps report more than 150 source-to sink pairs. The amount of source-to sink pairs has an average increase of 18 in an app. In this experiment, we demonstrate that Fragment's lifecycle has an effect on the data leakage detection result.

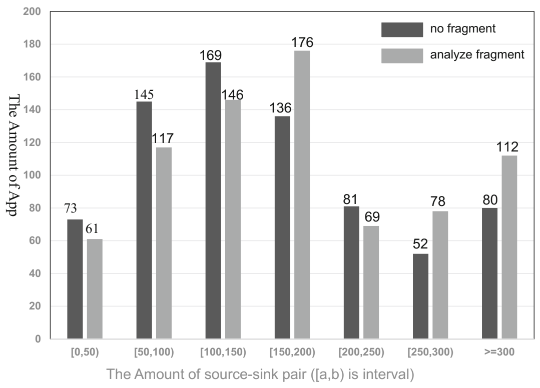


Fig. 10. Distribution of source-sink pairs

5.4 Runtime Performance

The runtime performance of FragDroid is shown in Figs. 11 and 12. The experiment result of the first app data set is depicted in Fig. 11. As is shown in this figure, when we do not analyze Fragment, 90% of the apps can be finished in 20 s and the average time is 12 s. After analyzing Fragment, 80% of the apps can be finished in 20ṡ and the average time is 14 s. It means that after we modelled the Fragment's lifecycle, the average overhead is 17%. The experiment result for popular app data set is shown in Fig. 12. After Fragment analysis, the overhead is 114%. The run time of analysis is highly depended on the amount of Fragments this app using.

6 Discussion

Current data flow analysis techniques on Android platform are not perfect. In this paper, we focus on fragment's lifecycle, and get a more complete control flow which is the prerequisite for data flow analysis. We have no in-depth analysis of native code and java reflection, so the data flow may be not precise enough. In addition, there exists a large number of callback methods in Android system,

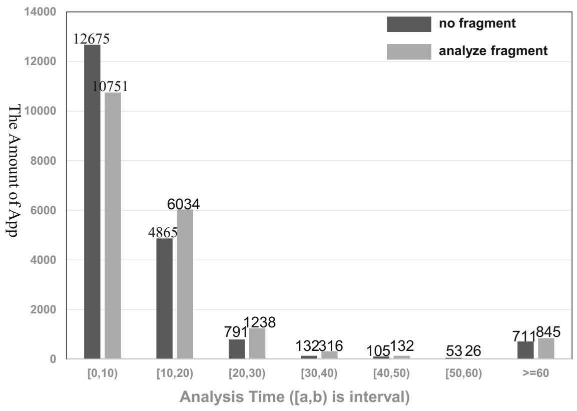


Fig. 11. Distribution of analysis time

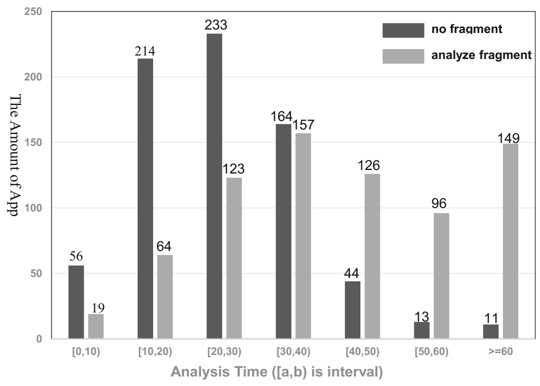
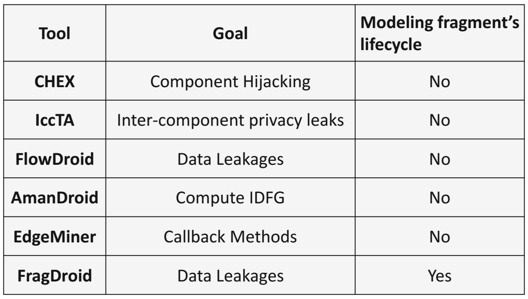


Fig. 12. Distribution of analysis time for popular apps

which can be rewritten to add malicious behavior. Thus, we just consider the control flow among callback methods. Actually, it is not complete, because these callback methods may have data dependencies. For example, in AsyncTask, the return value of doInBackground() is passed to onPostExecute() as the first parameter. But in our tool, we don't consider this. Malware can hide sensitive data flows through these dependencies. In the future, we can analyze the data dependence between callback methods' return values and parameters, and integrate these into data flow analysis procedure.

7 Related Work

Previous researchers have proposed some approaches to solve problems in static analysis on Android platform. We summarize the differences of existing static analysis tools in Fig. 13. CHEX [6], FlowDroid [7], AmanDroid [9] are three tools which perform static data flow analysis on Android platform. CHEX [6] is designed to detect the component hijacking problems in Android apps. When performing data flow analysis, CHEX analyzes each program split which includes code reachable from a single entry point at first. Cross split data flow are analyzed based on those system dependence graphs [24] which will be generated for every program split. FlowDroid [7] is aimed at detecting data leakages in Android apps. It models data flow analysis problem within the IFDS [25] framework for inter-procedural distributive subset problems. It also models the Activity's lifecycle in a dummy main method. AmanDroid [9] is an Android data flow analysis framework. It computes an inter-component data flow graph (IDFG) which contains all objects points-to information in both flow and context-sensitive way. IccTA [10] makes a more complete analysis on Android inter-component communication. It can decide the implicit intents' target, which can be Activity or Service. EdgeMiner [11] focuses on Android's callback. EdgeMiner designs an automatic approach to find callback methods on Android platform. These callback methods can be adopted to complement other data flow analysis tools like FlowDroid. None of the tools above considers Fragment's lifecycle, which may lead to the overlook of some data flows. Thus, in this paper we model the Fragment's lifecycle.



Tool	Goal	Modeling fragment's lifecycle
CHEX	Component Hijacking	No
IccTA	Inter-component privacy leaks	No
FlowDroid	Data Leakages	No
AmanDroid	Compute IDFG	No
EdgeMiner	Callback Methods	No
FragDroid	Data Leakages	Yes

Fig. 13. Comparison of different taint analysis tools

There are also some tools analyzing apps dynamically. TaintDroid [3] is one of them to modify the Dalvik virtual machine. Every instruction is interpreted by Dalvik, so TaintDroid can record the data flow relationship between objects. DroidScope [4] is an emulation based Android malware analysis engine that can be used to analyze Java and native components of Android Applications. It performs taint analysis on native instruction and dalvik instruction, so it's more precise than TaintDroid. SMV-HUNTER [13] is a tool designed to identify apps which is vulnerable to SSL/TLS Man-in-the-Middle attacks. AppAudit [14] is an efficient program analysis tool that detects data leakages in mobile applications. It combines static and dynamic analysis to overcome the shortcomings of each individual analysis.

Malware detection is an important topic in Android security research. RiskRanker [15], TriggerMetric [28] and DroidRanger [16] are heuristic-based malware detection tools. RiskRanker determines a malicious app according to risk behavior is performing in the app. TriggerMetric captures the static dependence relations between user inputs and sensitive operations providing critical system functions in programs. DroidRanger analyzes the permissions that malware and benign apps apply, then it identifies the combination of permissions which are frequently used in malware and rarely used by benign apps. Drebin [21], DroidAPIMiner [26], DroidMiner [22], DroidSIFT [23] and DR-Droid [27] identify malware based on machine learning algorithm. Drebin and DroidAPIMiner extract permissions and security APIs an app using to construct feature vector. DroidMiner uses control flow, while DroidSIFT uses data dependence. DR-Droid proposed a new Android repackaged malware detection technique based on code heterogeneity analysis, and the features in DR-Droid are extracted from each dependence region to profile both benign and malicious dependence region behaviors.

8 Conclusion

In this paper, we describe how Fragment's lifecycle can influence the data flow analysis result and propose an approach to model Fragment's lifecycle. To model the Fragment's lifecycle and its relationship with Activity's lifecycle, we design a tool FragDroid to generate a dummy main method which can model the control flow transfer between Fragment's and Activity's lifecycle methods. Our tool is built based on FlowDroid [7]. We perform some experiments using apps crawled from some alternative app markets. Experiments show that 8% of the selected apps use Fragment, and for most popular apps, the probability is 50.8%. We also evaluate our tool with the same data sets, the result shows the average overhead is 17%.

Acknowledgments. We would like to thank the anonymous reviewers for their comments. This work was supported in part by grants from the Chinese National Natural Science Foundation (61272078, 61073027, 90818022, and 61321491), and the Chinese National 863 High-Tech Program (2011AA01A202).

References

1. iOS and Android capture combined 98.4% share of smartphone market. http://www.macrumors.com/2016/02/18/ios-android-market-share-q4-15-gartner/
2. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE, May 2012
3. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 5 (2014)

4. Yan, L.K., Yin, H.: Droidscope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. Presented as Part of the 21st USENIX Security Symposium (USENIX Security 2012), pp. 569–584 (2012)
5. Vidas, T., Christin, N.: Evading Android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 447–458. ACM, June 2014
6. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting Android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 229–240. ACM, October 2012
7. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 29. ACM, June 2014
8. Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Highly precise taint analysis for Android applications. EC SPRIDE, TU Darmstadt, Technical report (2013)
9. Wei, F., Roy, S., Ou, X.: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of Android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1329–1341. ACM, November 2014
10. Li, L., Bartel, A., Bissyand, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: IccTA: detecting inter-component privacy leaks in Android apps. In: Proceedings of the 37th International Conference on Software Engineering, vol. 1, pp. 280–291. IEEE Press, May 2015
11. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: Automatically detecting implicit control flow transitions through the Android framework. In: NDSS (2015)
12. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61 (1995)
13. Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., Khan, L.: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014) (2014)
14. Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X.: Effective real-time Android application auditing. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 899–914. IEEE, May 2015
15. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 281–294. ACM, June 2012
16. Liang, S., Du, X.: Permission-combination-based scheme for Android mobile malware detection. In: 2014 IEEE International Conference on Communications (ICC), pp. 2301–2306. IEEE, June 2014
17. Baidu Android market. http://shouji.baidu.com/software/
18. Xiaomi Android market. http://app.mi.com/
19. Anzhi Android market. http://www.anzhi.com/
20. Android malware genome project. http://www.malgenomeproject.org/
21. Arp, D., Spreitzenbarth, M., Hbner, M., Gascon, H., Rieck, K., Siemens, C.E.R.T.: Drebin: effective and explainable detection of Android malware in your pocket. In: Proceedings of NDSS, February 2014

22. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: DroidMiner: automated mining and characterization of fine-grained malicious behaviors in Android applications. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8712, pp. 163–182. Springer, Cham (2014). doi:10.1007/978-3-319-11203-9_10
23. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware Android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1105–1116. ACM, November 2014
24. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(1), 26–60 (1990)
25. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61. ACM, January 1995
26. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICSSITE, vol. 127, pp. 86–103. Springer, Cham (2013). doi:10.1007/978-3-319-04283-1_6
27. Elish, K.O., Shu, X., Yao, D., Ryder, B., Jiang, X.: Profiling user-trigger dependence for Android malware detection. Comput. Secur. (C&S) **49**, 255–273 (2015)
28. Tian, K., Yao, D., Ryder, B., Tan, G.: Analysis of code heterogeneity for high-precision classification of repackaged malware. In: Proceedings of Mobile Security Technologies (MoST), in Conjunction with the IEEE Symposium on Security and Privacy, San Jose, CA, May 2016