Characterizing and Identifying Misexposed Activities in Android Applications*

Jiwei Yan
Tech. Center of Softw. Eng.
Institute of Software, CAS, China
Beijing, China
yanjw@ios.ac.cn

Xi Deng
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

Ping Wang
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

Tianyong Wu
State Key Lab. of Computer Science
Institute of Software, CAS, China
Beijing, China

Jun Yan †
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

Jian Zhang†
State Key Lab. of Computer Science
Institute of Software, CAS, China
Univ. of Chinese Academy of Sciences
Beijing, China

ABSTRACT

Exported Activity (EA), a kind of activities in Android apps that can be launched by external components, is one of the most important inter-component communication (ICC) mechanisms to realize the interaction and cooperation among multiple apps. Existing works have pointed out that, once exposed, an activity will be vulnerable to malicious ICC attacks, such as permission leakage attack. Unfortunately, it is observed that a considerable number of activities in commercial apps are exposed inadvertently, while few works have studied the necessity and reasonability of such exposure. This work takes the first step to systematically study the exposing behavior of EAs through analyzing 13,873 Android apps. It utilizes the EA associated call relationships extracted from byte-code via data-flow analysis, as well as the launch conditions obtained from the manifest files, to guide the study on the usage and misexposure of EAs. The empirical findings are that the EA mechanism is widely adopted in development and the activities are liable to be misexposed due to the developers' misunderstanding or carelessness. Further study on subsets of apps selected according to different criteria indicates that the misexposed EAs have specific characteristics, which are manually summarized into six typical misuse patterns. As a consequence, ten heuristics are designed to decide whether an activity should be exposed or not and are implemented into an automatic tool called *Mist*. Experiments on the collected apps show that around one fifth EAs are unnecessarily exposed and there are more than one third EAs whose exposure may not be suggested.

*This work is supported by National Natural Science Foundation of China (Grant No. 61672505), the National Key Basic Research (973) Program of China (Grant No. 2014CB340701), and Key Research Program of Frontier Sciences, CAS, Grant No. QYZDJ-SSW-JSC036.
†Corresponding Authors. Email: yanjun@ios.ac.cn, zj@ios.ac.cn

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238164

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software testing and debugging**;

KEYWORDS

Android apps, Exported Activity, Program Analysis

ACM Reference Format:
Jiwei Yan, Xi Deng, Ping Wang, Tianyong Wu, Jun Yan, and Jian Zhang. 2018. Characterizing and Identifying Misexposed Activities in Android Applications. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238164

1 INTRODUCTION

With the growing momentum of the Android OS and its app market, the functionalities of apps become rich and specialized. It is a trend that more and more apps share information and collaborate with each other to complete a complex task. For instance, an electronic-payment app can be invoked by multiple third-party e-commerce apps to perform the payment process. Another example is that many apps are embedded with social contact functionality by invoking some mature social contact apps, e.g., *Facebook* and *WeChat*.

By default, an activity in an app is invisible to the external apps, i.e., they can not be activated by activities in other apps. The Android system provides the "Exported Activity" (EA for short) mechanism by which an app can share specific activities with other apps. EAs can be regarded as the interfaces of apps, which usually carry the key functionalities that the developers want to promote and will be repeatedly invoked by other apps. Besides, the invocation to EAs may be indeterministic due to the Android specific mapping mechanism (e.g., implicit invocation, see Section 2.3), which brings flexibility and uncontrollability to their usage.

Many activities are made to be exposed (i.e., EA) in practice. According to our statistics, around two thirds of apps have at least one EA, and the percentage of EA in all activities is about 8.6% (see Section 4.1). Existing works [1–3] have shown the vulnerability of EAs. For example, the arbitrary data received from external apps may disturb the apps' regular work-flow or even cause code injection,

and the invoking of sensitive APIs in EA without the protection of permissions may cause permission leakage. A recent research [4] also shows that inter-component communication (ICC) methods are significantly used by malware to leak private data. Thus, we have the following hypothesis: **exposed activities might bring the potential vulnerabilities to the entire application**. Although the EA simplifies inter-app interaction, new potential security risks and functionality bugs that it might bring force developers to think carefully before using it.

Recent works [5–9] have provided a series of effective defect detection approaches that involve EA, however, little is known about whether the exposing behavior of an activity is proper or not. The wide adoption of such a flexible and error-prone EA mechanism drives us to re-examine the activities that are declared to be exposed. Thus, we make a large-scale empirical study on 12,673 commercial apps and 1200 open-source ones in order to answer the following three research questions:

- **RQ1 (Usage of EA):** To what extent are EAs used by developers? Which functionalities are EAs mostly used to perform?
- **RQ2 (Comparative Analysis):** Are there any differences between the well exposed activities and those should not necessarily be exposed?
- **RQ3 (Characteristics of Misexposed EA):** Are there any common patterns of unnecessarily exposed activities? Can these misexposed EAs be identified automatically?

To answer these questions, we first obtain the launch conditions of EAs and extract intents that can start certain EAs via data-flow analysis on apks. Then, we match intents to activities according to the Android mechanism and construct the caller-callee pairs within the app set. We further employ a comparative analysis on two sets of selected apps to investigate the misexposure of activities. The comparison results indicate that the misexposed EAs have specific characteristics, which are manually summarized into six typical misuse patterns. Furthermore, we sum up ten heuristic rules based on these patterns to categorize the EAs into four classes, and implement these ten rules into a tool called *Mist* for automatic misexposed EA identification. The experiments on the collected apps show that about 19.23% EAs are unnecessarily exposed and there are 36.50% EAs whose exposure may not be appropriate.

To sum up, the contributions of this work lie in three-fold:

(1) We conduct an empirical study on three datasets to understand the usage and misexposure of EAs in practice. To the best of our knowledge, this is the first research that systematically studies the behavior of activity exposure.
(2) We make use of static analysis techniques to extract EA related information from tens of thousands of real-world apps to aid the misexposure identification.
(3) We summarize six kinds of misuse patterns and extract ten rules, based on which a tool called *Mist* is designed and implemented for EA misexposure identification. The experiments show that our tool can effectively help to find the misexposed EAs. Both our tool and the related data are publicly available on GitHub (https://github.com/AndroidMist/Mist).

2 BACKGROUND

This section provides the background knowledge about Android system and exported activity.

2.1 Android Activity

Android is an open-source and Linux-based operating system developed by Google for portable devices. Except for some native libraries, Android apps are mainly written in Java and compiled into Dalvik byte-code, while they also have some configuration files (e.g., manifest file) to declare the components and layouts. Android apps are composed of four kinds of components, including *Activity*, *Service*, *Content Provider*, and *Broadcast Receiver*. Among them, activity is the most frequently used component that provides an interface for the users' interaction. A research [4] shows that around 67.4% of the total ICC method calls are related to the activity component. The activity can be categorized into two kinds, including the *Internal Activity (IA)* and *Exported Activity (EA)*. The former one can only be launched by the components in the same app, while the latter one allows other Android apps to launch it. The EA provides an effective way for the ICC among multiple apps, which is the major component discussed in this paper.

2.2 Activity Attribute and Intent Filter

Android system provides a number of attributes for the activity [10], which can be mainly set in the manifest file by developers. We only list the relevant ones here.

- `android:exported` By default, this attribute is set as `false`, i.e., the activity is an IA. Developers can change it to exported by setting its value as `true`.
- `android:permission` This attribute defines the permission which is required for external apps to activate this activity. If the caller app does not declare this permission, it is not allowed to call this activity.

The `intent filter` is an element of EA that is also set in the manifest file, while an EA can have multiple intent filters. It decides what kinds of implicit intents the EA responds to. An intent filter contains several sub-elements, including `action`, `category`, and `data`. The resolution from intent to intent filters is performed on these elements based on specific rules [11]. If multiple intent filters are compatible, the system displays a dialog showing options for users to pick up which app to start. The intent filter has an attribute `android:priority` that provides information about how able an activity is to respond to the intent that matches the filter. Android will consider only those with higher priority values as potential targets for the intent.

2.3 Activity Exposing and Launching

There are two rules to expose an activity, according to the Android reference [10]. First, if the attribute `android:exported` is set as `true`, this activity will be an EA. Besides, for an activity whose attribute `android:exported` is not set, it will also be an EA if it contains at least one intent filter [12]. Note that, only effective intent filters are considered, e.g., intent filters with no action are ignored because they can not be invoked. The rest activities are internal ones (IAs).

Both EA and IA can be launched by the mechanism of intent [11], which can be categorized into two types, *Explicit Intent* and *Implicit Intent*. The explicit intents specify its target by giving the activity name (the fully-qualified class name), while the implicit intents only declare a series of features (e.g., action and category), which makes invoking components of other apps possible without the knowledge of the concrete component name [11]. Android system provides a set of APIs to assign these attributes that are related to activity launching, some of which are listed in Table 1. The combination of these attributes determines the type of invocation (explicit or implicit) and which activity to be launched.

Table 1: some attribute assignment APIs of Intent

Intent Attribute	API
component	setClass, setClassName, setComponent, Intent(Context, Class)
action	setAction, Intent(String)
category	addCategory
data	setData
action, data	Intent(String, Uri)
action, data, component	Intent(String, Uri, Context, Class)

When receiving an explicit intent, the activity with the same component name will be directly picked up for reaction. For implicit intents, the system needs to find appropriate activities to start by comparing descriptions of the intent and the intent filters of other activities. It first compares the action information between the intent and all intent filters of EAs. The action specified in the intent must match one of the actions listed in the filter. For category matching, every category in the intent must match a category in the filter. For intent without category, the category `android.intent.category.DEFAULT` will be added by default. Thus, each intent should declare the default category, or else it will not be launched by any implicit intent. For the data element, each part of it is a separate attribute in the intent filter, which can be composed as a uri: `<scheme>://<host>:<port>/<path>` and then be used for data matching.

2.4 An Example of EA

In this subsection, we illustrate the EA and intent mechanism by a simple example. Figure 1 shows a portion of manifest file of an EA `FooActivity`. This EA declares `exported=true` explicitly as well as a permission named `com.intent.permission.Foo`. In addition, it has an intent filter for implicit intent calls. This intent filter will respond to intents that satisfy the following three conditions: containing action `com.intent.action.Foo`; having no category or only the default one; containing data element whose corresponding uri matches `http://foo`.

Figure 2 shows an activity that can launch this EA. It creates an intent instance with a string as the parameter, which denotes that the action is `com.intent.action.Foo`. Then it sets the category and data attributes, where the value of `category` is obtained by invoking the method `getCategoryStr()` and the value of `data` is obtained from the parameter `uriData`. Note that if the default category attribute is not set in the code, the Android system will add it to the intent by default. Then the data items are attached, in which `bundle` is a mapping from `String` keys to various values.

```
<activity android:name="FooActivity"
  android:exported="true"
  android:permission="com.intent.permission.Foo" >
  <intent-filter>
    <action android:name="com.intent.action.Foo"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="http" android:host="foo"/>
  </intent-filter>
</activity>
```

Figure 1: Manifest File of FooActivity

```
public class FooLaunchActivity extends Activity {
  public void startWithData(String uriData) {
    Intent intent = new Intent ("com.intent.action.Foo");
    String myCategory = getCategoryStr();
    intent.addCategory(myCategory);
    intent.setData(Uri.parse(uriData));
    intent.putExtra("key1", "abcdefg");
    intent.putExtra("key2", new Bundle());
    bundle.putInt("key3", 12345678);
    this.startActivity(intent);
  }
}
```

Figure 2: Activity for Launching FooActivity

Finally, this intent will be performed with the API `startActivity` and be resolved to the intent filter of the activity `FooActivity`, which will be launched by the Android system after resolving.

3 EMPIRICAL STUDY METHODOLOGY

To the best of our knowledge, there are no systematic empirical researches that investigate the usage and exposing characteristics of EA mechanism in Android apps. Thus, we conduct an empirical study to investigate how EAs are used in real-world Android apps. In this section, we introduce the experimental dataset and the method of the empirical study.

3.1 Analysis Method

To systematically study the usage of EA, we perform a statistic analysis to obtain the EA declaration information in manifest, including the most commonly used actions and the exposing modes of EAs, etc. And to make clear how EAs are invoked in the program, we analyze the intent related APIs by applying a light-weight data-flow analysis on bytecode of the app. Then we construct two databases based on the analyzing results, i.e., the *EA declaration database* and *EA invocation database*. The first one stores the EA declaration information, including activity attributes as well as intent filters, and the second one stores all possible invocations used in the program. Finally, the information in the two datasets is matched according to the Android mapping mechanism to construct the caller-callee pairs within the app set, which can guide the study on the usage and misexposure of EAs. The overall analysis process is shown in Figure 3.

3.2 Dataset Collection

There are three datasets in our empirical study, in which Dataset AL contains all the apps we collected from app markets, and the other two are selected from Dataset AL according to different criteria.

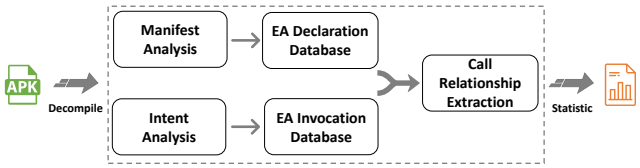


Figure 3: Overall Analysis Process

Dataset AL: 13,873 Android apps from three markets. They are collected from three app markets, including an open-source app repository F-Droid (1,200, 10.8%), Google Play (6,946, 50.1%), and a Chinese app market Wandoujia (5,727, 41.3%). For F-Droid, we download all the available apps in June 2017 using a web crawler. Considering the scale of Google Play and Wandoujia, obtaining all their apps is not possible. Since Google Play has 32 categories and Wandoujia has 14 ones, we download the first 500 apps of each category by the display order. For Google Play, direct app downloading is not allowed, thus we get apps from a third-party website APKLeecher [13] and miss some apps. Besides, apps that can not be decompiled are excluded from the dataset. Figure 4 displays the downloads and size distributions of the collected apps using box-plots. As we can see, these Android apps are widely used. Among them, half have their download counts in the range from one hundred thousand to one million, as demonstrated by the solid median line. And the size of the apps ranging from hundreds of KB to hundreds of MB, is at the average of 15MB in Google Play and 18MB in Wandoujia, as shown by the X mark symbol.

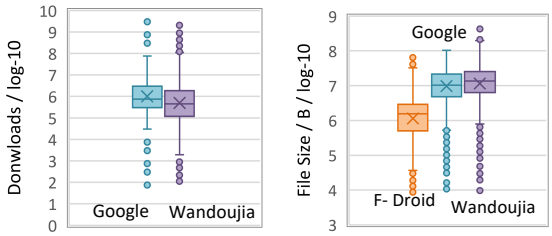


Figure 4: Downloads and File Size Distribution.

Dataset AR: Apps containing EAs that might not necessarily serve as external interfaces. We focus on the apps that have different characteristics with most others and suppose that EAs in them might not necessarily be exposed to external apps. EAs are specially designed for external interaction rather than internal main functionality. According to our statistic, most apps expose only a small part of activities for external invocation, however, the percentage of EA in some apps are abnormally high. To get knowledge about the generation of this dataset, we make a small empirical study. And in this study, we take the ratio of EA against the number of activities as the indicator for abnormal app identification. First, we pick the top 5 apps with the most abnormal ratio of EAs from Dataset AL and extract 327 EAs from them. By launching each EA and inspect its bytecode to judge the reasonableness of the exposure, the tester report that 234 EAs (71.6%) are suspected to be misexposed. Thus, in our experiments, we distinguish the outlier apps that have the abnormal ratio of EAs in Section 4.1 as Dataset AR based on the following assumption: **the apps with an abnormal ratio of EAs compared with most of the apps are more likely poorly programmed.**

Dataset MD: Apps containing EAs that are more likely well declared. To make a comparative analysis between normal apps and outlier apps in Dataset AR, we want to extract EAs that have a higher possibility of being well declared. Considering that widely used apps are more likely to be developed under strict code regulations by skilled programmers and might have been well tested, the EAs corresponding to such apps are more likely to be well declared and suitable for comparison. We also make a small empirical study for the generation of this dataset. We pick 5 apps with the most downloads from Dataset AL and extract 47 EAs from them. By launching each EA and inspect its bytecode to judge the reasonableness of the exposure, we find that only 5 EAs (10.6%) are labeled as misexposed activities. And most of (4/5) these widely used apps provide specific SDKs (e.g., WeChat's open SDK[14]) for external invoking so that have only a small percentage EA (15/476 for WeChat). Therefore, we pick up 50 apps that have the most downloads in Dataset AL to generate Dataset MD.

3.3 Manifest Analysis

According to the Android reference, any activity must be declared in the manifest file, which is written in XML language. Therefore, the complete EA declaration report can be obtained by parsing and analyzing manifest files. We first decompile an apk file into two parts, the manifest file and the Dalvik byte-code. Then, we identify all the ⟨*activity*⟩ tags, and collect the activity attribute (e.g., `android:exported`) as well as the intent filter (e.g., `android:action`) information.

3.4 Intent Analysis

To have insights into the activity invocation, we make a static analysis of ICC messages on the program byte-code. In this step, we make use of a Java bytecode analysis framework *soot* [15] for data-flow analysis. Considering that existing intent analysis tool is inefficient (e.g., 140 seconds per app using IC3) in analyzing our large-scale datasets, we adopt a light-weight intent analysis method in this paper. Our method is mainly based on the reaching definition technique [16], which focuses on statically determining which definitions may reach a given point in the code.

As shown in Algorithm 1, we perform a variable assignment analysis on ICC messages. For each method under analysis (in line 2), we first construct use-define chains [17] to capture the data propagation. In line 3, we recognize all the invocation units by locating instructions that invoke API `startActivityForResult` or `startActivity`, and add them into set *invs*. Then, in line 5, we make use of the use-def chain to obtain the corresponding intent object `intent` for each unit in *invs*. Next, we collect all the units that are related to object `intent` and form a set `uses` in line 6. In line 7-13, we iterate over all the units in set `uses` aiming to find out all the assignments of each intent-related attribute. Considering that each API listed in Table 1 can be used, we first get the name of attribute (e.g., `attr_name=action`) using `getAttrName(use)`.

For this attribute, we get a mapping of (attribute, variable) by invoking method `getAttrVar(udc, use)`, and then get the assignments of each variable, i.e., a (variable, assignments) mapping, by invoking `getVarAssign(method, var)`. Finally, the obtained information will be added to the output set by invoking method `storeIntentInfo` in line 12.

To get the corresponding assignment of a variable, one difficulty is that not all assignments of it can be directly obtained. For a variable assigned as the return value of a function call, we search the callee method by its `methodSignature` and get the last assignment of the return value. And for a variable assigned in the caller method that is passed as a parameter, we search the caller method by its name through call graph. Because there may be several caller methods, we locate all the call points and add these corresponding assignments into a candidate set. Besides, we take care of branch statements and get all the possible assignments according to the use-def chain. The assignment obtaining algorithm of method *getVarAssign*(*method*, *var*) is shown in Algorithm 2.

3.5 Call Relationship Extraction

Android system has a mechanism for mapping an intent invocation to activities. When an intent is invoked for activity launching explicitly, the Android system will wake up the target activity by its component name. However, for the implicitly invoked intents, the Android system needs to search all apps for intent filters that match the given intent. We simulate the matching process (refer to Section 2.3) between the caller and callee to extract call relationships for the apps we collected. Then, we can obtain the matching results on a specific dataset, e.g., the caller set of given EA or the callee set of an intent object.

4 OBSERVATIONS FROM EMPIRICAL STUDY

In this section, we discuss our major findings by previous analyses. All of our analysis processes are performed on an Intel Xeon CPU @2.40 GHz machine, with 64 GB memory and Ubuntu 16.04 operating system. It takes about 33 hours to decompile all the apk files (8 seconds per app), 34 minutes for manifest file analysis and around 80 hours for intent analysis (20 seconds per app).

Algorithm 1 Activity Invocation Analysis

```
Input: method name method
Output: intent info
1:  res_list = ∅
2:  udc = getUseDefineChains(method)
3:  invs = getAllUnitsInvokingAPI("startActivity")
4:  for each unit inv in set invs do
5:      intent = getIntentVariable(udc, inv)
6:      uses = getUseUnitSet(udc, intent)
7:      for each unit use in set uses do
8:          if unit use assigns attribute attr then
9:              attr_name = getAttrName(use)
10:             attr_var = getAttrVar(udc, use)
11:             attr_assign = getVarAssign(method, attr_var)
12:             storeIntentInfo(intent, attr_name, attr_assign)
13:         end if
14:     end for
15: end for
```

Algorithm 2 *getVarAssign*(*method*, *var*)

```
Input: method, var
Output: assignment
1:  udc = getUseDefineChains(method)
2:  unit = getDefUnit(var)
3:  if unit is caller method then
4:      callee = getCalleeMethod(unit)
5:      var = getRetVar(callee)
6:      assignment = getVarAssign(callee, var)
7:  else if unit is callee method then
8:      callers = getCallerMethods(unit, getCallGraph())
9:      for each method caller in callers do
10:         var = getParameterVar(caller, method, para_index)
11:         assignment = getVarAssign(caller, var)
12:     end for
13: else
14:     assignment = getUnitAssign(unit)
15: end if
16: return assignment
```

4.1 EA usage

In this part, we discuss the usage and functionality of EA.

Percentage of EA. To figure out whether EA is frequently used in real-world apps or not, we count the number of activities and EAs declared in each app. The detailed results of the apps collected from different app markets are shown in Table 2, where the second to fourth columns show the number of apps (#N), activities (#A) and EAs (#EA), and the fifth column shows the number of apps that have at least one EA (#AEA). Note that a special kind of EA, `MainActivity` (whose action is `android.intent.action.MAIN` and category is `android.intent.category.LAUNCHER`), is excluded in the process of counting, since each `MainActivity` is the entry of its app and is EA by default. Totally, we get 639,483 activities, among which 55,075 activities are EAs (8.6%). And there are 9,361 (67.5%) apps that have at least one EA (#AEA), which indicates that **the EA mechanism is widely adopted in the real-world Android applications**.

Table 2: Number of Activities and EAs

App Market	#N	#A	#EA	#AEA
F-Droid	1200	6898	1109	492
Google Play	6946	195973	17514	4243
Wandoujia	5727	436612	36452	4626
Total	13873	639483	55075	9361
App Market	**#Avg$_A$**	**#Avg$_{EA}$**	**#EA/#A**	**#AEA/#N**
F-Droid	5.7	0.9	16.0%	41.0%
Google Play	28.2	2.5	8.9%	61.1%
Wandoujia	76.2	6.4	8.3%	80.1%
Average	46.1	4.0	8.6%	67.5%

We calculate the ratio of EAs (#EA/#A) of all the collected apps and display the results in Figure 5. The apps collected from different markets are labeled with different colors and shapes. These apps also have different characteristics, e.g., open-source apps in F-Droid always contain fewer activities than commercial ones. For commercial apps, the apps in Wandoujia are with a higher ratio of EAs and are more dispersed than these in Google Play. We have two observations from Figure 5: **1) some apps have extremely higher percentage of EAs than others; 2) when the number of activities increases, the ratio of EAs decreases in most cases**.

The *Local Outlier Factor* (LOF) algorithm [18], is used to identify outliers based on the density. Although apps with many activities

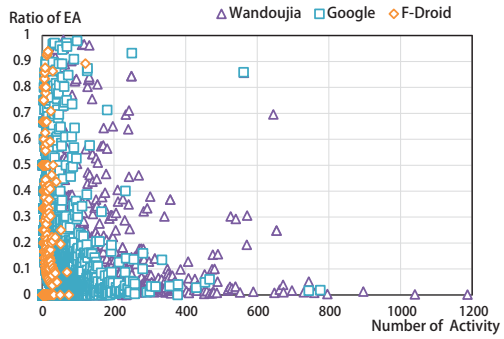


Figure 5: Distribution of the Ratio of EAs

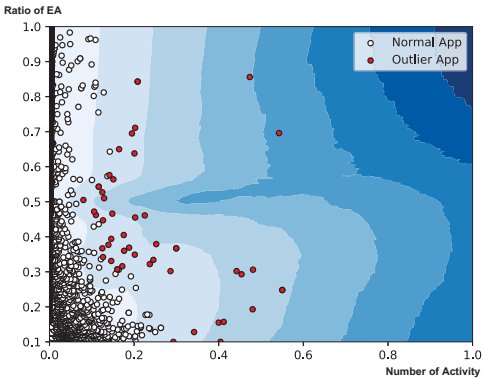


Figure 6: Outlier App Detection Result

and few EAs might also be taken as outliers by the LOF algorithm, they are more likely to be well-programmed apps indeed. Therefore, we filter out the apps whose ratio of EAs is less than 0.1. The final results are displayed in Figure 6, in which the background color denotes the contour plot based on density. The lighter the color, the higher the density. To maintain the number consistency of two datasets, the top 50 outliers (red points) are labeled as *outlier apps* and added into Dataset AR.

Functionality of EA. As introduced in Section 2.3, an EA may contain several intent filters to declare what kinds of implicit intent the EA responds to. The action attribute in the intent filter indicates which kind of operations the EA will perform as well as shows its main functionality. So we extract all the intent filters from the manifest file and get 63,758 actions that are in 10,010 types. Among these actions, 44,181 (69.3%) actions are system ones that are defined in the `Intent.java` file of Android source code.

Table 3: Most frequently used actions

	Actions Declared in Manifest
system	android.intent.action.VIEW (58.6%)
	android.intent.action.SEND (3.1%)
	android.intent.action.SEARCH (1.8%)
	android.appwidget.action.APPWIDGET_CONFIGURE (1.2%)
	android.intent.action.CREATE_SHORTCUT (1.0%)
non-system	com.sina.weibo.sdk.action.ACTION_SDK_REQ_ACTIVITY (2.5%)
	com.google.android.gms.appinvite.ACTION_PREVIEW (0.9%)
	cn.jpush.android.ui.PushActivity (0.3%)
	COM_TAOBAO_TAE_SDK_TRADE_WEB_VIEW_ACTION (0.2%)
	com.google.android.gms.actions.SEARCH_ACTION (0.2%)

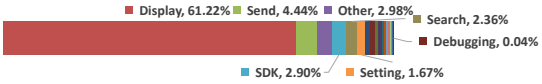


Figure 7: Functionality of Actions.

Table 3 shows the most frequently used actions in our EA declaration database. The most frequently used system action is `android.intent.action.VIEW` (58.6%), which is used to display the data to users, e.g., text or webpage browsing. And the most frequently used non-system action is `com.sina.weibo.sdk.action.ACTION_SDK_REQ_ACTIVITY` (2.5%), which is used to callback the caller app who invokes *sina weibo* (a social networking application) with a sharing operation. We also find that **the frequently used non-system actions are always provided by giant companies**.

To make clear which functionalities are mostly used by developers, we pick the 300 most used types of actions (more than 80% of count) declared in the manifest and manually analyze their functionalities. We get 21 categories in total and the results are shown in Figure 7. The categories not shown are Shortcut, Pick, Stub, Camera, Nfc, Push, Communication, Media, Launch, Hardware, Auth, Create, Share and Install. As we can see, **EA can support various functionalities, in which display (61.22%) is the most commonly used functionality.** One interesting finding is that actions related to SDK account for 2.90%, which are provided by big companies like *sina*, *taobao*, and etc. By providing SDK to developers, these companies can protect and manage the invocation of their EAs easily. Another observation is that the debugging and testing related EAs still exist in some released apps, which may bring potential security issues and should not be exposed in the published version.

4.2 Comparative Analysis

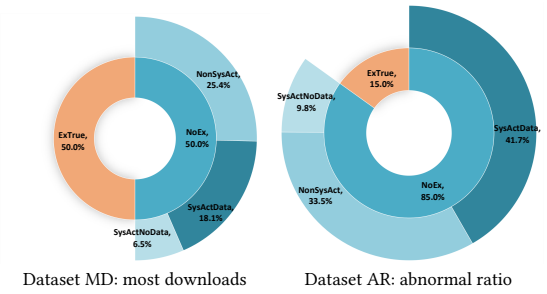
We now get two small datasets using different picking criteria, in which Dataset MD contains EAs that belong to widely used apps and Dataset AR contains EAs that come from apps with abnormal ratio of EAs. Each dataset contains 50 apps, and we find that these two datasets are disjoint.

Table 4: General Information of Datasets

	App_Num	App_Size	EA_Num	EA_Ratio
Dataset MD	50	1,332MB	598	8.1%
Dataset AR	50	1,522MB	5654	38.5%

Table 4 lists the general information about the two datasets. As we can see, the first difference between these two sets is the number of EAs. While the size is similar, apps in Dataset AR have much more EAs than those in Dataset MD. The last column gives the ratio of EAs, which shows EAs in Dataset AR have a larger proportion.

We further investigate how developers expose an EA with different exposure modes and show the results in Figure 8. The exposure mode "ExTrue" indicates the EAs whose attribute `exported=true` are explicitly declared, and "NoEx" indicates the EAs without that attribute. The ratio of activities declared with `exported=true` varies a lot in these two datasets, which is 50% on Dataset MD but only 15% on Dataset AR. It shows that **the attribute `exported`, which demonstrates the intention of developers explicitly, is more**



Dataset MD: most downloads Dataset AR: abnormal ratio

Figure 8: Exposure Mode Comparison

often used in well programmed apps. For EAs in mode "NoEx", we further separate them into three types "SysActData", "SysActNoData" and "NonSysAct", according to whether they contain any data and non-system action or not. The results show that **the mode "SysActNoData" is rarely used in both datasets, which might be an abnormal exposure mode**.

4.3 Misexposure Patterns of EA

The results of the comparative analysis guide the misexposure pattern extraction. Besides, manual inspection on the 100 apps from Dataset MD and AR as well as the Android reference also helps. We find six misexposure patterns and separate them according to whether developers know the exposing characteristic or not.

4.3.1 Developer-unanticipated Exposure. As the activity exposing mode is flexible, the developers might not know that their activities are exposed.

P1: Abnormally High Percentage of EA. ***1) How to Extract:*** The comparative results in table 4 show that poorly programmed apps may have higher EA percentage, i.e., declare more misexposed EAs. By comparing the ratio of EAs in each app, we find that some apps have extremely high EA numbers and ratios, whose functionalities are usually not designed for external invocation. ***2) Case Study:*** The app *Mobile collaboration*, which exceeds millions of downloads in the market, contains 59 activities and 58 of them are EAs. Its ratio of EAs reaches 98.3%, while the average value is 8.6%. This app is designed for mobile teamwork cooperation, and all functions must be accessed by users who have logged in. However, we observe that these EAs can be easily accessed through the direct external invocations without login, that may violate the intention of developers. Another example is the well-known app *Bing Dictionary*, which has totally 63 activities and 38 of them are EAs. However, according to the name of these EAs and the judgment by manual invocation, most of them seem to be part of the internal functionalities instead of being specially designed for external invocations. ***3) How We Identify:*** This pattern can be identified by EA ratio statistic.

P2: Copy-Pasted EA Declaration. ***1) How to Extract:*** When developers want to declare an activity correctly, the most convenient way is to imitate the last declared one, i.e., declare by copy-and-paste. By manual inspection on the manifest files of the selected apps in both datasets, we find that copy-pasted EA declarations widely exist. ***2) Case Study:*** We find an app called *ToolWiz Photos*, which has totally 197 activities and 128 of them are EAs. Surprisingly, up to 124 EAs in this app are declared using mode

```
<activity android:exported="true" android:name="com.toolwiz.
photo.community.UserInfoActivity" />
<activity android:exported="true" android:name="com.btows.
photo.editor.ui.SelectiveColorActivity"/>
```

Figure 9: Copy-Pasted EA Declaration

"ExTrue". In Figure 9, we show some of them and get rid of all the expose-irrelevant attributes. If we start activity `UserInfoActivity` using adb command [1], the app will crash and throw an exception, which means that the developers do not expose it deliberately. And `SelectiveColorActivity` is used for adjusting the tone for a photo. External invocation is allowed to directly start it without a target image, which leads this activity to be invalid and even causes a crash. ***3) How We Identify:*** Misexposures in this pattern can be found out by calculating the ratio of each mode (see Section 4.2) in one app.

P3: Inappropriate Action and Data. ***1) How to Extract:*** According to the Android reference and manual inspection, we find that the officially provided system actions (or categories) are commonly used, which are difficult to be taken as the identifier of an EA. Developers always add data, a uri object that assigns the data to be acted on, to limit the range of resolved activities. Therefore, the EA declaration that contains only system actions without data item required is likely to be misused. ***2) Case Study:*** To ease understanding, we discuss a real case in *UCMobile*. As shown in Figure 10, it declares an EA with the most frequently used system action `android.intent.action.VIEW` only. When an implicit call that only contains this system action is sent, dozens of EAs are matched as candidates to be launched. ***3) How We Identify:*** This pattern can be identified by analyzing the intent-filers of each EA.

```
<activity android:name="com.ucweb.activity.LifeAssistantActivity">
  <intent-filter>
     <action android:name="android.intent.action.VIEW"/>
     <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

Figure 10: Inappropriate Action and Data

P4: Incorrect Category Setting. ***1) How to Extract:*** According to the official reference, Android system will automatically apply the category `android.intent.category.DEFAULT` to all the implicit intents, so that an EA must include such category in its intent filters. Otherwise, no implicit intents will resolve to this EA with incomplete intent filters. However, they are still exposed in force and can be invoked by explicit intents. ***2) Case Study:*** As shown in Figure 11, app *Mozilla Browser* declares such an EA that contains an action only. This activity fails to be invoked implicitly, however, it is exposed by declaring intent-filter and can be invoked explicitly. When this EA is invoked explicitly, a blank window will show up and then disappear. Then the logcat will give the error log `android.view.WindowLeaked`, which means the activity has a leaked window. Obviously, this activity is not ready to be exposed as an EA. ***3) How We Identify:*** We identify this pattern by detecting the existence of the default category.

[1] Considering that the lack of extra data will also cause app crash, all EAs we listed in this section do not need to receive any extra data.

```
<activity android:name="org.mozilla.gecko.sync.config.activities.
SelectEnginesActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
  </intent-filter>
</activity>
```

Figure 11: Incorrect Category Setting

P5: Implicit internal Invocation. ***1) How to Extract:*** In the light of the Android reference [10], activities declared with intent-filters will expose an activity. Developers should prevent other apps from calling one activity by not setting intent filters for it. However, by our investigation, intent-filters are often used for implicit intra-invocation due to its convenience, rather than external invocation. The comparative results in the previous section also show that widely used apps declare EA in "ExTrue" more frequently. ***2) Case Study:*** The app "dianping" contains 483 EAs and 481 of them are in the mode "NoEx". Some of them are used for implicit invocation inferred by the name of these activities, e.g., "MyCardActivity" and "AtFriendActivity". ***3) How We Identify:*** We identify this pattern by comparing the ratio of each exposure mode and provide warning information to developers.

4.3.2 Developer-anticipated Exposure. There are also some EAs that are exposed for the convenience of development and should be removed before application being released, otherwise they may bring unpredictable threats.

P6: Debugging Functionality. ***1) How to Extract:*** In our manual inspection, we find that some activities are designed and exposed to ease the debugging, mainly according to their names. These exposed debugging activities help to test the target activity directly, and they may manipulate the database and lack security protection, Forgetting to remove them in the release versions might be an issue. ***2) Case Study:*** For instance, the invoking of the EA `DebugDomainSelectActivity` in Figure 12 launches an activity that is used for *domain testing*. ***3) How We Identify:*** By keyword retrieving, we totally find 13 debugging activities in this app.

```
<activity android:name="com.dianping.debug.DebugDomainSelectActivity" >
  <intent-filter>
    <action android:name="com.dianping.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

Figure 12: Debugging Functionality

5 MISEXPOSURE CHARACTERIZATION AND IDENTIFICATION

Through the investigation of collected datasets, we summarize a series of rules to identify whether an EA is misexposed or not. First, we give several features of activity in Table 5, including information about EA declaration and invocation as well as the attributes used in the identification of the misexposure patterns. Each feature corresponds to a boolean variable and the second column describes the condition to make it true.

Then we summarize the characteristics of misexposed activities according to the patterns in Section 4.3, and give out the classification rules in Table 6. As shown in the first column, EAs can be classified into four classes according to the necessity and reasonability of the exposure. All the rules can be categorized into two types, i.e., "Must" and "May", in which the type "May" contains some coarse-grained rules that can not tell which specific EA is misexposed. For example, the rule `similar` can only figure out a series of similar EAs to be suspicious. The second column shows the priority (*Pr*) of rules, which is set according to our experience. The last column describes the judgment conditions, which uses single or the combination of features listed in Table 5. Because one EA may satisfy several conditions at the same time, when condition collision occurs, the final classification is determined by the priority value. For example, the rule `debug` that maps to class "MustIA" has high priority; while another rule `exTrue` (mayEA) is more undetermined. When both these conditions are satisfied by one EA, it will be classified as "MustIA" at last.

Then we use a logic programming language *prolog* to automatically identify the misexposures, in which the program logic is expressed in terms of relations, represented as *facts* and *rules* [19, 20]. A fact is composed of an attribute and its value, and a rule is in the form of `Head:-Body.`, in which the *Head* is the conclusion and the *Body* contains several facts. If the facts in Body are true, the Head is true.

Figure 13 shows part of our implementation using *prolog*, in which the declaring order of rules decides the priority of matching. We use features listed in Table 5 as attributes of fact, whose values can be extracted through EA declaration and invocation analysis. For each EA, we can get totally ten facts to help the classification. And we make use of conditions and their classes in Table 6 to define

Table 5: Features of EA

Feature	Description
`exTrue`	declares `exported=true`
`ifTrue`	contains `intent filter`
`noDefault`	omits the default category
`sysActNoData`	declare only system action without data
`priority`	contains priority setting of intent filter
`permission`	contains permission setting of activity
`clsDeclare`	with classname that has been declared more than three times in manifest
`clsInvoke`	with classname that has been externally invoked
`actInvoke`	with non-system action that has been externally invoked
`similar`	belongs to an app that declared EA with the similar exposure mode shown in Figure 8, including ExTrue, SysActData, SysActNoData and NonSysAct.
`debug`	contains keywords "test", "debug", etc.
`highRatio`	belongs to an app that has high value of #EA/#A

Table 6: The Classification Conditions of EA

Class	Pr	Condition
MustEA	4	`clsInvoke` or `actInvoke`
	5	`clsDeclare`
	6	`priority` or `permission`
MayEA	9	`exTrue`
MayIA	7	`similar` (P2)
	8	`highRatio` (P1)
	10	`ifTrue` and not `exTrue` (P5)
MustIA	1	`sysActNoData` and `ifTrue` and not `exTrue` (P3)
	2	`noDefault` and `ifTrue` and not `exTrue` (P4)
	3	`debug` (P6)

ten rules, in which the IA classification conditions are linked to misexposure patterns. For example, line 4-5 in Figure 13 represent a rule, which means if the fact `clsDeclare(true)` is satisfied, the class of corresponding activity is "mustEA". For other rules that contain several facts, these facts are combined, where the comma indicates "and", and the semicolon indicates "or". For instance, the lines 1-3 mean that if an EA satisfies `(noDefault(true) and not ifTrue(true) and not exTrue(true)) or (debug(true))`, it belongs to class "mustIA".

6 EVALUATION

We describe some experiments in this section. A tool called *Mist* (MISexposure idenTification for Android) is designed and implemented based on the extracted rules. For the coarse-grained rule `highRatio`, we find out the apps that have more than 50 EAs and a ratio of EAs larger than 0.4. For rule `similar`, we only detect apps that have more than 30 EAs and use thresholds that vary from 0.5 to 0.7 for different exposure modes. Then we perform experiments on a large-scale dataset and two small ones, which aims at answering the following two research questions:

- **RQ4 (Usefulness):** To what extent are our misexposure identification results consistent with manual checking? Does the identification help developers to reduce ICC attack threats?
- **RQ5 (Result Distribution):** What is the distribution of EAs misexposed in real-world Android apps? Do the results vary in different datasets?

6.1 RQ4: Usefulness

We randomly select 50 apps and launch their EAs to verify the identification consistency between *Mist* and manual checking. Besides `MainActivity`, EAs that need to receive `extra data` are also excluded, since the uncertain launching results induced by different inputs will obstruct the manual judgment. Totally, we obtain 519 EAs from the selected apps.

As we know, there are no common criteria to determine whether an activity should be exposed or not, and no existing benchmark is provided for this task. Therefore, to get the correct classification results for tool evaluation, we design the following regulations for manual identification according to the launching result:

- The activity is an IA if it has data dependencies with other unexecuted internal activities and thus provides incomplete functionalities; shows abnormal display (e.g., blank window); throws exceptions; or provides obviously internal functionalities (e.g., debugging).
- The activity is an EA if it provides complete functionalities for external apps.

For each of the EAs, we automatically generate an adb launching command, such as `adb shell am start -n package/activity -d data`, according to the EA declaration. The options `-a` (for action) and `-c` (for category) are not adopted to ensure only one target will respond. When the target EA is launched, the GUI interfaces and the exception information from logcat are considered in manual annotation, but the code information is not provided.

```
activity(mustIA):-
  noDefault(true), not(ifTrue(true)), not(exTrue(true)); debug(true);
activity(mustEA):-
  clsDeclare(true).
```

Figure 13: Part of the Implementation Using Prolog

All the 519 EAs are labeled by three annotators as EA or IA in about five hours, while the automatic identification by `Mist` only takes several seconds. We compare the results committed by all annotators and find that they annotate same labels on 367 EAs. i.e., all of them label an activity as EA/IA, and the disagreement among annotators is 29.7%. It is hard for annotators to infer the intention of developers from the decompiled byte-code, thus, annotators may have disagreement on the same activity. For example, some EAs receive uri as the data item, which can be composed as `<scheme>://<host>:<port>/<path>`. It is easy to generate a legal input but it may not make sense, which may cause invalid invocation or blank window. The launching of these EAs is difficult to judge by annotators who are not the developers of the app under test. Therefore, we perform a strict selection, i.e., we just consider the activities that are classified into the same classes by all annotators as the test oracle and drop the rest. The final results show that 263 (71.6%) EAs are successfully identified as the same type by the annotators and our tool. The precision and recall of the misexposure identification are 0.87 and 0.77, respectively. For those inconsistent ones, the rules `exTrue` and `similar` hit the most ones (over 50% in total), which should be further studied and refined.

We also collect some apps that have been reported as victims of ICC attacks. For example, Covert [6] is a popular permission leakage detection tool, whose results are publicly available on website [21]. There are totally ten activities that are reported as victims of the permission leakage by Covert, in which seven of them are `MainActivity`. We identify the rest three activities using *Mist* and find out two of them seem to be misexposed. By manual inspection, these two activities (from two apps) are inferred to be misexposed ones (since they are designed for implicit invocation rather than external invocation), which indicates that the vulnerabilities of these apps can be fixed by unexposing their activities.

6.2 RQ5: Result Distribution

We extract and collect features for all EAs in apps of the three datasets, and make use of the rules written in *Prolog* to classify these EAs. Then we obtain the classification results and show their statistics in Figure 14.

As we can see, in Dataset AL, there are 19.23% of EAs that are classified as "mustIA" with high certainty. Overall, more than half (55.73%) of EAs are suspected to be IAs, whose exposure may not be suggested. Our further analysis shows that, in the 9361 apps which

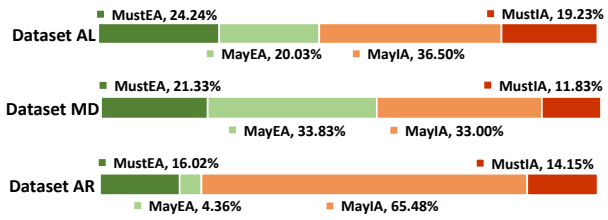


Figure 14: Identification Results on Three Datasets

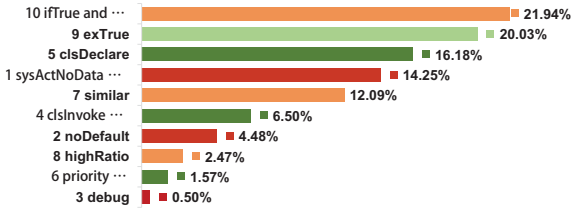


Figure 15: Statistic Results of Each Rule

have at least one EA, 5339 (57.03%) of them are detected to have at least one misexposed EA. These results indicate that **the misexposure of EA widely exists in real-world apps**. For the other two datasets, there are 55.16% of EAs seem to be correctly exposed in Dataset MD. And for the Dataset AR, only 20.38% of EAs seem to be rightly exposed, which is consistent with our assumption.

To have some intuitive understanding of our rules, we count the hitting number of each rule on Dataset AL. As we can see from Figure 15, the rule `ifTrue and not exTrue` hits the most EAs. These EAs are suspected to be IAs and should be checked by their developers. Rule `exTrue` also holds for a large proportion, which shows that a number of developers show their exposing intention explicitly. And about 16.18% of EAs satisfy the rule `clsDeclare`, which may be public activities provided as the SDK interfaces. For other rules, they hit fewer EAs even some of them have high priority, indicating that they are discriminative for identification.

7 THREATS TO VALIDITY

As with any system, *Mist* has its limitations, including the scale of benchmarks, the setting, and the research scope. First, the identification results are limited by the scale of our dataset, especially when the rules that involve the number of invocations are used. Besides, both the threshold in each rule and the priority value to combine these rules are set manually, which may limit the accuracy of identification. These settings can be obtained by a combined technique of program analysis and machine learning. Finally, Android apps are composed of four kinds of components, and only the most commonly used component `activity` is studied in this paper. We believe that the misexposure of other components has similar characteristics with activity and will investigate them in our following studies.

8 RELATED WORK

ICC Attacks Detection. Currently, most of the existing works on testing and analysis of Android apps [22–27] target at the intra-app analysis. However, the design of ICC has its limitations, which may cause bugs or security flaws. A recent study by Ahmad et al. [28] discussed the challenges it brings to Android development. Chin et al. [1] provide tool ComDroid to describe application communication vulnerabilities caused by the misunderstanding of the intent passing system, e.g., unauthorized intent receipt and intent spoofing. The research [2] proposes an iterative test generation approach to detect the ICC vulnerabilities (e.g., XSS, SQL injection, etc.) of Android apps. In each iteration, they recovered the custom fields (variables) of intent by instrumenting the APIs that are used to read such fields and monitoring the app execution. Bagheri et al. [6] implement a tool Covert that can detect the permission leakage caused by the lack of permission requirements of exposed components. They first perform the static analysis techniques to obtain the model of program behavior, and then use the alloy language (an object modeling notation) to model the combination of apps, and finally perform the formal analysis technique to verify the model. In addition to a wide variety of approaches to identifying vulnerabilities, an exploit generation tool LetterBom [7] based on a combined path-sensitive symbolic execution-based static analysis is provided, which can be used to reduce the number of false positives in vulnerability detection. In our work, we pay more attention to another aspect, i.e., detect whether activities should be exposed or not instead of detecting its vulnerabilities.

Intent Analysis. The implicit control flow introduced by ICC mechanism makes the generation of precise call graph and control flow graph, which are the essential parts of program analysis, very difficult. In recent years, several researchers aim to expose such implicit transitions by intent analysis [29–31], in which the tool Epicc [29] is provided by Octeau et al. for obtaining the ICC methods and their parameters. They also provided a tool IC3 [30] (Epicc has now been replaced with IC3 [32]) which modeled the ICC messages with proposed COAL language and implemented the associated solver that performs a string analysis to figure out the ICC specification in Android apps. Based on Epicc and IC3, Li et al. [4] developed IccTA, a static analysis tool for detecting inter-component privacy leaks in Android apps. The links between the components are detected by the code instrumentation and static analysis techniques. Besides the activity, Zhang et al. [33] focus on service testing by extracting service related intents using a variable assignment analysis.

9 CONCLUSION

In this paper, we investigate the exposing behavior of EAs, which is rarely discussed in existing works. Since EAs usually carry the functionalities that are eagerly promoted by developers and vulnerable to malicious ICC attacks, eliminating the unnecessarily exposed EAs is a simple but effective way to improve the quality of Android apps. Therefore, the key challenge lies in the identification of the misexposure. With the help of the static analysis, we derive typical misexposed activities from tens of thousands of real-world apps. By investigating these activities, we summarize the specific characteristics of misexposed EAs into typical misuse patterns, which in general are related to the misunderstanding and carelessness of developers. We also design several heuristics for misexposed EA identification and implement a tool called *Mist*. The experiments on real-world apps show that it can effectively help to locate the misexposed EAs, and the problem of the misexposure of activities is widespread and noteworthy.

Our tool can improve the quality and robustness of Android applications by detecting misexposures of activities. In the future, we will improve the accuracy of our tool by using more concise static analysis and employing machine learning techniques to obtain the weights (including the thresholds and the priority) from a training set with sufficiently labeled samples. We will also make further studies on the exploitation of the misexposed activities as well as how to re-implement the EA declarations automatically to fix the misexposures .

REFERENCES

[1] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David A. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011)*, pages 239–252, 2011.

[2] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[3] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David A. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS*, pages 627–638, 2011.

[4] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, pages 280–291, 2015.

[5] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *19th Annual Network and Distributed System Security Symposium, NDSS*, 2012.

[6] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. COVERT: compositional analysis of Android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, 2015.

[7] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 661–671, 2017.

[8] Jun Ma, Shaocong Liu, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. Lesdroid - a tool for detecting exported service leaks of Android applications. The preprint is available at website http://moon.nju.edu.cn/people/junma/static/files/LesDroid(pre-print).pdf, 2018.

[9] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 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*, pages 1329–1341, 2014.

[10] activity | Android Developers. https://developer.android.com/guide/topics/manifest/activity-element.html, 2017.

[11] Intents and Intent Filters | Android Developers. https://developer.android.com/guide/components/intents-filters.html, 2017.

[12] intent filter | Android Developers. https://developer.android.com/guide/topics/manifest/intent-filter-element.html, 2017.

[13] Online APK Downloader | Download APK Directly From Google Play To Your Computer. http://apkleecher.com/, 2017.

[14] Open SDK. https://open.weixin.qq.com/cgi-bin/showdocument?action=dir_list&t=resource/res_list&verify=1&id=1417751808&token=&lang=en_US, 2017.

[15] Soot. http://www.bodden.de/2008/09/22/soot-intra, 2017.

[16] Reaching definition | Wikipedia. https://en.wikipedia.org/wiki/Reaching_definition, 2017.

[17] Use-define chain - Wikipedia. https://en.wikipedia.org/wiki/Use-define_chain, 2017.

[18] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 93–104, 2000.

[19] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[20] Dennis Merritt. *Building expert systems in Prolog*. Springer Science & Business Media, 2012.

[21] covert. http://www.ics.uci.edu/~seal/projects/covert/index.html.

[22] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 29:1–29:11, 2014.

[23] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of Android applications in droidsafe. In *22nd Annual Network and Distributed System Security Symposium*, 2015.

[24] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117, 2015.

[25] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, pages 426–436, 2015.

[26] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of Android malware based on the usage of data flow APIs and machine learning. *Information & Software Technology*, 75:17–25, 2016.

[27] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *37th IEEE/ACM International Conference on Software Engineering*, pages 303–313, 2015.

[28] Waqar Ahmad, Christian Kästner, Joshua Sunshine, and Jonathan Aldrich. Inter-app communication in Android: developer challenges. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016*, pages 177–188, 2016.

[29] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proceedings of the 22th USENIX Security Symposium*, pages 543–558, 2013.

[30] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering*, pages 77–88, 2015.

[31] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple Android apps to support inter-app analysis. In *Proceedings of 30th International Conference on ICT Systems Security and Privacy Protection*, pages 513–527, 2015.

[32] Epicc. http://siis.cse.psu.edu/epicc/.

[33] Li Lyna Zhang, Chieh-Jan Mike Liang, Yunxin Liu, and Enhong Chen. Systematically testing background services of mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 4–15, 2017.