A Flexible Intent Fuzzer with an Automatic Tally of Failures for Robustness of Inter-Component Communication in Android Apps

Kwanghoon Choi

Chonnam National University, Gwangju, Republic of Korea

Myungpil Ko

Yonsei University, Wonju, Republic of Korea

Byeong-Mo Chang*

Sookmyung Women's University, Seoul, Republic of Korea

Abstract

Context: Android apps are based on a component-based structure with inter-component communication (ICC) mechanism called *Intent*. They have a robustness problem due to the mishandling of malformed ICC, which is called *Intent vulnerability*.
Objective: This research aims at a new flexible Intent fuzzer with an automatic tally of failures for detecting Intent vulnerabilities of Android apps causing the robustness problem.
Method: This paper proposes two new ideas. First, we design an Intent specification language to describe the structure of Intent, allowing our Intent fuzz testing tool to be flexible by having a well-defined interface between Intent generators and Intent executors. Second, we propose a tally method automatically classifying unique failures determined by the longest common subsequence algorithm. We have implemented an Intent fuzz testing tool called *Hwacha* with the two ideas, and have evaluated it with 50 commercial Android apps thoroughly.
Results: Our Intent fuzz testing tool offers an arbitrary combination of Intent generators and executors due to the presence of a well-defined interface between them. The automatic tally method excluded almost 80% of duplicate failures in our experiment, reducing efforts of testers very much in review of failures. The tool uncovered more than 400 unique failures, including a new one unknown so far. We also measured execution time for Intent fuzz testing, which has been rarely reported before. Finally, we give a survey on Intent fuzz testing for robustness of Android apps.
Conclusion: The proposed Intent fuzz testing tool has a flexible structure in combining generators of Intent test cases with arbitrary executors, and it automatically performs the whole procedure of Intent vulnerability testing including failure classification. These two features are applicable to all the existing Intent fuzz testing tools without difficulties.

Keywords: Android, Fuzz, Robustness, Exception, Mobile phone

1. Introduction

Android is one of dominant computing platforms today because it has the leading mobile market in smart phones and it is open source software. The number of Android apps running on the platform reaches

*Corresponding author
Email addresses: kwanghoon.choi@jnu.ac.kr (Kwanghoon Choi), myungpil.ko@yonsei.ac.kr (Myungpil Ko), chang@sookmyung.ac.kr (Byeong-Mo Chang)

2.81 millions as of December 2016. Many people in the world use these Android apps on a daily basis. A software with such a large user base needs to be very robust and secure, otherwise even a small number of defects may lead to significant costs.

However, Android has been known to be vulnerable because of a few reasons. One is because there are many Android platform versions coexisting in the market from the newest version to the old ones. Another is because of the well-known *fragmentation problem* of compatibility that stems from the nature of open-source software. The third reason is the loosely coupled component based structure of Android apps, which is good at increasing modularity and reuse but needs to be used very carefully. This paper focuses on the third aspect: a robustness problem caused by malformed Inter-component communication in Android apps.

Most Android programs are written in Java with Android APIs (Application Programming Interfaces). Android is Google's open-source platform for mobile devices, and it provides the APIs necessary to develop applications for the platform in Java [1]. An Android program consists of components such as Activity, Service, and Broadcast Receiver, where the components communicate among themselves by sending messages called *Intents*.

Due to Intents that may miss any fields or may carry ill-formed values, Android components are vulnerable. Let us consider an example of Android Activity component *Note* in Figure 1. This Activity component constitutes a mobile screen with windows such as text labels, displaying a title and a content given by an Intent with which this component is activated. A caller component should set these title and content strings in the Intent properly before it activates *Note* to invoke its *onCreate* method. Also, a caller component should specify an action in the Intent to request *Note* to perform. As shown in the code below, the *Note* Activity accepts two actions: *"INSERT"* for creating a new memo and *"EDIT"* for editing an existing one.

```
public class Note extends Activity {
  String title;
  String content;
  void onCreate(Bundle savedInstanceState) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if(action.equals("android.intent.action.EDIT")){
       title   = intent.getStringExtra("title");
       content = intent.getStringExtra("content");
    } else
    if(action.equals("android.intent.action.INSERT")){
       title   = "Title";
       content = "Type your memo";
    }
    // Display a title and a content
  }
}
```

Figure 1: Android Program Example

There are four kinds of *Intent vulnerability* in the example. First, if an action is missing, it is set as NULL and then the first occurrence of invocation of *equals* will throw `NullPointerException`. Second, if an action other than these two is specified, displaying the title and content will cause to throw `NullPointerException`: both of the tests on the kind of actions in the two `if` statements evaluate to false, leaving title and content be `NULL`. Third, if an Intent with an action *"EDIT"* misses one of values of two keys "title" and "content", the same problem will happen because one of the invocations of *getStringExtra* returns `NULL` due to the missing value. Fourth, if any value of the two keys is set to be, say, an integer (not a string), an invocation of *getStringExtra* method with the corresponding key will return `NULL` because of the incorrect type of the value.

Recent researches [2, 3, 4, 5, 6, 7] have developed *Intent fuzzer*s to detect such Intent vulnerabilities.

Basically, they generate arbitrary (possibly malformed) Intents by their own strategies, and test if any running of Android app crashes on invocation with the Intents. It has been reported that they have uncovered many interesting Intent vulnerabilities in Android apps.

The structure of the existing Intent fuzzers, however, has two common problems. First, each Intent fuzzer has one's own Intent generation strategy. One cannot use the other Intent fuzzer's strategy without changing much of one's implementation. Null IntentFuzzer [2] sets only the null value to all Intent fields. JarJarBinks (JJB) [3] improved it with random and semi-valid Intents. DroidFuzzer [4] focused only on the data field of Intents with malformed audio and video files. In IntentFuzzer [5], IntentDroid [6], and ICCFuzzeer [7], static and dynamic analyses have been employed to construct Intents more relevant to what Android apps deal with.

Second, the researches on Intent fuzz testing seem to report few things about how to classify failures automatically in Android crash logs. Several different Intents can lead to the same failure, and so we should identify the multiple occurrences of the same failure in testing with the Intents. Due to the nature of fuzz testing, the number of Intents to test with tends to be large, and so the manual classification of failures in testing could be very time consuming. This problem becomes more serious under a situation such as in Android Marketplace that a manager would apply some Intent fuzz testing tool to more than millions of Android apps to filter out ones having Intent vulnerability and to report to their developers automatically.

In response to the two problems, we first propose an Intent fuzz testing tool, which clearly decouples Intent generation from execution with Intents. For this, we design *Intent specification language* as a flexible way to describe the structure of Intents. The proposed language offers a well-defined interface so that programmers or any tools can write arbitrary Intent specifications for a generator to produce Intents, which an executor takes for testing.

We next propose an automatic tally method for classifying failures using a traditional algorithm [8] on the longest common subsequence (LCS) problem. The similarity of two crash logs is defined by the ratio of the length of the LCS over the longer length of the two crash logs. This criterion can identify two failures resulting from the same exception in the same program point, allowing some minor differences such as thread IDs that are numbered differently per each run.

Finally, based on the two new ideas, we have implemented a fully automatic Intent fuzzer, which we believe is the first practical Android Intent fuzzer. We have performed an experiment over 50 commercial Android apps using the developed tool to automatically uncover more than 400 unique failures of Intent vulnerability causing crashes. We have analyzed the experiment results in detail to assess a state-of-the-art in the area of Intent fuzz testing.

The contributions of this paper are summarized as follows:

- We have designed an Intent specification language to describe the structure of Intent, which makes our Intent fuzz testing tool flexible allowing an arbitrary combination of Intent generators and Intent executors through a well-defined interface.

- We have also proposed a tally method automatically classifying failures determined by the LCS algorithm, reducing much efforts on manually analyzing Android crash logs to identify different failures.

- The two new ideas mentioned above are not for a particular tool, but they are universally applicable to all Intent fuzz testing tool for Android apps. Particularly, the idea of using Intent specification language can be a basis for the existing tools to cooperate with another.

- We have implemented a fully automatic Intent fuzz testing tool, and have demonstrated the effectiveness of the two ideas by applying it to 50 commercial Android apps and finding many unique Intent vulnerabilities including one unknown so far.

- Our Intent fuzz testing tool itself is designed to be robust, which is important in practice particularly when many Android apps are tested. This allowed us to report time for Intent fuzz testing that has been rarely evaluated before.

Section 2 presents motivation for Android Intent vulnerability. Section 3 introduces our proposal on Intent specification language. In Section 4, we describe an Intent fuzzer tool in detail using the Intent specification language and the Android crash log grouping method. In Section 5, we show an experiment results using our Intent fuzzer tool with commercial Android binary apps. In Section 6, we give a comprehensive survey of a state-of-the-art on Intent fuzz testing for Android applications. In Section 7, we compare our tool with the existing ones. Section 8 concludes.

2. Motivation: Intent Vulnerability

An Android program generally forms a Java program with APIs in Android platform. Using the APIs, one can build user interfaces to make a phone call, play a game, and so on. An Android program consists of components such as Activity, Service, Broadcast Receiver, or Content Provider. Activity is a foreground process equipped with windows such as buttons and text inputs. Service is responsible for controlling background jobs, and so it has no user interface. Broadcast Receiver reacts to system-wide events such as notifying low power battery or SMS arrival. Content Provider is an interface of various kinds of storage including mobile database systems.

Components in an Android program interact with each other by sending messages called *Intent* in Android platform. An Intent holds information about a target component to which it will be delivered, and it may hold data together. For example, a user interface screen provided by an Activity changes to another by sending an Intent to Android platform, which will launch a new screen displayed by a target Activity specified in the Intent.

The use of Intent is advantageous for reuse of Android components by making them be loosely coupled with others. For example, Android platform provides popular mobile services such as making a phone call, sending an SMS, and using a web browser by Activity components, and many Android apps easily make use of them just by sending some Intent to the platform. Programmers deal with Android components outside an Android app in the same manner as they do with those inside it.

However, many misuses of Intent have been reported to cause Android apps crash in [2, 3, 4, 5, 6, 7], called *Intent vulnerability*. The misuses of Intent are typically due to malformed Intent examples with NULL action, invalid actions, missing extra data, and ill-typed extra data, as discussed in the introduction. There is an analogy between sending an Intent and a function invocation, as follows:

Sending an Intent		Function Invocation	
Elements	Error Types	Elements	Error Types
Action	Unknown action	Function	Function not defined
# of extra data	Missing extra data	# of args	Missing arguments
Types of extra data	Ill-typed extra data	Types of args	Ill-typed arguments

Although the problem of Intent vulnerability is serious, there is no good mechanism yet as for function invocation, to issue a warning for Intent vulnerability in Android apps. Programmers should chase after causes of Intent vulnerability in Android apps with much efforts. Unfortunately, Android programmers write a piece of code for component activation by Intent in the form difficult to uncover the potential Intent vulnerability. For example, to invoke Note Activity component in the example, we write as follows:

```
Intent i = new Intent();
i.setTarget("com.example.android.Note");
i.setAction("android.intent.action.EDIT");
i.setExtraString("title", "... my title ...");
i.setExtraString("content", "... my content ...");
startActivity(i);
```

where an action name, arguments, and argument types are scattered over several statements and so it is not so easy to verify correctly the validity of the whole of an Intent to invoke Note Activity. As a result, Intent vulnerability in the code remains silent in compile-time, and then it arises in runtime. This makes it

```
INTENTSPEC ::= { FIELD FIELD ... FIELD }
             | { FIELD FIELD ... FIELD } || INTENTSPEC

FIELD ::= COMPONENT | ACTION | DATA | EXTRA | CATEGORY | TYPE | FLAG | INTERNAL

COMPONENT ::= cmp = COMPTYPE COMPNAME
COMPTYPE  ::= Activity | Service | BroadcastReceiver | ContentProvider
COMPNAME  ::= ID / (ID | .ID)
ACTION    ::= act = ID
DATA      ::= dat = URI
EXTRA     ::= [ ID=EXTRAVALUE , ... , ID=EXTRAVALUE ]
CATEGORY  ::= cat = [ ID , ... , ID ]
TYPE      ::= typ = URI
FLAG      ::= flg= [ ID , ... , ID ]
INTERNAL  ::= internal = BOOL

EXTRAVALUE ::= String STRING? | boolean BOOL? | int INT?
             | long LONG? | float FLOAT? | uri URI? | component COMPNAME?
             | int[] INTARRAY? | long[] LONGARRAY? | float[] FLOATARRAY?

INTARRAY   ::= INT , ... , INT
LONGARRAY  ::= LONG , ... , LONG
FLOATARRAY ::= FLOAT , ... , FLOAT

LETTER ::= (A - Z | a - z)+
ID     ::= LETTER (A - Z | a - z | 0 - 9 | _ | . | $)*
URI    ::= LETTER (A - Z | a - z | 0 - 9 | _ | . | / | : | * | ? | @ )*

(BOOL, INT, LONG, FLOAT, and STRING denote the corresponding primitive values.)
```

Figure 2: A Syntax for the Intent Specification Language

difficult to identify misuses of any Intents in Android apps early. Another reason of difficulty comes from *implicit Intent* with no target component specified. One can verify validity of Intents only after a target component for the Intents is determined. This is a serious problem of the Android ICC (Inter-component communication) design.

We therefore approach this problem of Intent vulnerability by Intent fuzz testing. This paper proposes a new Intent fuzzer which is flexible and provides an automatic tally of failures for detecting the explained Intent vulnerabilities in Android apps.

3. The Intent Specification Language

Before we go to an exposition on our Intent fuzz testing tool, this section introduces a specification language for describing the structure of Android Intent, which is one of important elements in the tool.

Let us begin with an example of Intent specification for the structure of Intents that Note Activity is supposed to receive in Figure 1, written in the proposed language, as follows:

```
{ cmp = Activity com.example.android/.Note
  act = android.intent.action.EDIT
        [ title = String, content = String ] }
|| { cmp = Activity com.example.android/.Note
     act = android.intent.action.INSERT }
```

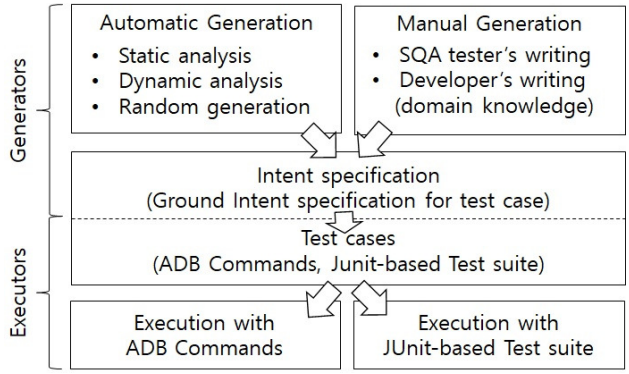

Figure 3: Decoupling Generators from Executors by Intent Specification Language

An Intent specification is a sequence of fields surrounded by { and }, and it can be built by a disjunction (||) of two Intent specifications as in the example above. A sequence of fields denotes a set of Intents satisfying the description of the fields. In the first sequence of the example, *cmp* is a field name bound to a target Activity component, com.example.android.Note class, and *act* is bound to an action (name), "android.intent.action.EDIT". Fields for extra data to perform an action with are surrounded by [and], and each extra data field describes a key and an associated type information, such as *title* and *String*. The second sequence of fields in the example denotes another set of Intents similarly but posing no constraints on other than those for cmp and act. By interpreting || as the set union, the example of Intent specification denotes the union of the two sets of Intents. Every Intent specification can thus be regarded as a predicate that defines a set of Intents.

Note that every instance of Intent can be neatly presented by a special form of Intent specification. We call it a *ground Intent specification*. Every field in a ground Intent specification is assigned a value, and there is no field that declares only types. A ground Intent specification is interpreted as a single Intent only with the specified fields and with no extra fields. A particular test case of Intent for our fuzz testing is described by this ground Intent specification form.

```
{ cmp = Activity com.example.android/.Note
  act = android.intent.action.EDIT
        [ title   = String "my title" ,
          content = String "your content" ]
  dat = URI http://our.uri.com }
```

The full detail of the Intent specification language is shown in Figure 2. We design the Intent specification language by modeling the structure of Intent class in Android platform [1] including a target component (cmp), an action (act) with data (dat) and type (typ), a list of extra data (each of which is a tuple of a key, a type of a value, and an optional value). For an exposition on category and flag, readers may refer to the developer's document available in [1].

The proposed language extends the structural information of Intent with the disjunction of field sequences (||) to express many alternative forms of Intents for testing. It also specifies component type information (COMPTYPE) such as Activity because the ways of testing execution varies depending on the component types. It allows us to describe a field type information such as integer, string, arrays, uris (URI) and so on. It can also specify together concrete values for the field.

The Intent specification language allows a modular structure of Intent fuzz testing tool by decoupling generators of Intent test cases from the associated executors, which is very flexible in combining a generator with an executor for the testing in many ways, as depicted in Figure 3. This structure enables a machine to generate automatically specifications for Intents Android component communication, or it offers a way of writing them manually to a human such as SQA testers and developers. The automatic generation of Intent specification makes use of static analysis and dynamic analysis with random generation. Some static

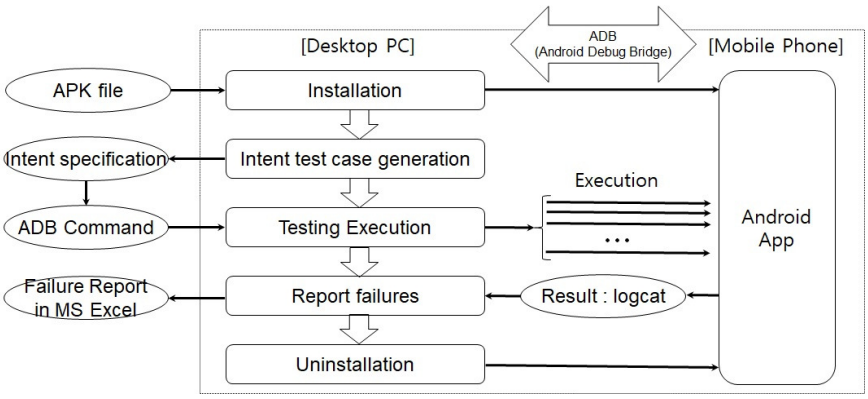

Figure 4: The Architecture of Our Intent Fuzzer

analysis discovers the potential structure of Intents communicated among components in an Android app [5, 7], and some dynamic analysis monitors the execution of the Android components to capture the usage of values in Intents [6]. A random generation strategy can be employed to fill in insufficient information in the generated Intent specifications.

SQA testers or developers can write Intent specifications manually based on their domain knowledge on what Intents a target Android component expects to take. Such domain knowledge can originate from specifications for an Android project.

As well as the generator side, the Intent specification language enriches the executor side. Given a (ground) Intent specification, no matter what generator is involved in generating the specification, one can choose an executor option for one's own purpose. Every ground Intent specification is directly mapped onto ADB (Android Debug Bridge) commands or a JUnit-based Android test suite, both of which are immediately executable for Intent fuzz testing.

Our Intent fuzzer offers all combinations of automatic and manual generation of Intent test cases with two executors using ADB commands and JUnit-based test suite, as described in Figure 3. A detailed usage of our tool for manual writing Intent specification and for generating and executing JUnit-based test suite can be found in a companion web site [9]. In the following section, we will focus on a combination of automatic generation and ADB command based execution.

4. A Flexible Intent Fuzzer with an Automatic Tally of Failures

We design a fully automatic Intent fuzz testing tool offering a flexible scheme by decoupling the Intent test case generation from the testing execution allowing arbitrary combinations of Intent generators and testing executors. Figure 4 describes the structure of the Intent fuzz testing tool starting with installation of each APK file on mobile phone and finishing with removing it. The tool runs on a desktop PC and is connected to an Android mobile phone for executing testing Android apps via ADB (Android Debug Bridge) interface provided by Android SDK [1]. The tool automatically constructs Intent specifications based on the component configuration of an input APK file, it maps them onto ADB commands to execute on the installed Android app, it receives the test execution logs via another Android SDK tool called *logcat* [1], and it automatically filters the duplicates of the logs to write a report in Microsoft Excel format for review.

4.1. Generating Intent Test Cases

The procedure of Intent test case generation takes a given APK file and generates ground Intent specifications by two steps. First, we identify target Android components and associated Intent specifications for testing from the component configuration information in the APK file. Second, we apply a fuzzing strategy to transform the generated Intent specifications into ground ones.

4.1.1. Construction of Intent Specification from Android Component Configuration

To explain this procedure, one needs to understand the component configuration of an APK file in detail. APK is a ZIP format file holding a configuration file called AndroidManifest.xml, a binary executable (named classes.dex), and resource files such as bitmap images. In the configuration file, a list of Android components in the APK is declared. Each declaration of an Android component names a Java class of the component. The declaration also provides information on Intents to activate the component with, which is called *Intent filter*. Android platform makes use of this class name and Intent filters to determine which Android component to activate. When an Intent holds a Java class name for a target Android component, it is called explicit Intent. Android platform attempts to pass every explicit Intent to the specified component unconditionally. When an Intent does not specify any target name of Android component, it is said to be implicit. Every implicit Intent is passed to an Android component only when it is matched with the Intent filter of the component. Android platform thus makes use of Intent filters to find a target Android component on a given implicit Intent. Therefore, Intent filter declarations in the component configuration file can be used to construct a skeleton of Intent specification automatically as in the following.

In this procedure, our Intent fuzzer tool uncompresses a given APK file to retrieve a list of Android component declarations in the configuration file, collecting Intent filters for each Android component declaration, and uses the filter information to generate minimal Intent specifications. For example, an APK file with Note Activity in the example of Figure 1 has a configuration file as shown in Figure 5. One of the declared components in the configuration file is Note Activity, and the Intent filter enumerates two action names, INSERT and EDIT.

```
<manifest ...  package="com.example.android" ...>
 ...
 <application ...>
  <activity android:name="com.example.android.Note">
     <intent-filter>
        <action android:name="android.intent.action.INSERT"/>
        <action android:name="android.intent.action.EDIT"/>
        ...
     </intent-filter>
  </activity>
  ...
 </application>
</manifest>
```

Figure 5: An Example of AndroidManifest.xml

Then our tool constructs an Intent specification from Intent filter information in Android component declarations retrieved from the APK file. For example, from the example configuration file above, it constructs:

```
{ cmp = Activity com.example.android/.Note
  act = android.intent.action.EDIT }
|| { cmp = Activity com.example.android/.Note
     act = android.intent.action.INSERT }
```

No Android configuration file declares valid key names and types for extra fields of Intents such as title and String, and so the Intent specification constructed above is missing this information. There are a few ways to fill in the extra field part of Intent specification. One can perform static or dynamic analysis to extract such extra field information from a target Android app automatically, and one can make use of one's domain knowledge on a target Android app to write it manually.

4.1.2. A Fuzzing Strategy for Generating Ground Intent Specification

Once an Intent specification is constructed for each Android component, what to do next is to generate ground Intent specifications that represent executable Intent test cases such as ADB commands or JUnit-based test suite. No matter what method is employed for such a construction, we can safely assume that we are given Intent specifications written in the proposed language, describing the characteristics of Intent test cases of interest. In the following, we will discuss a fuzzing strategy for refining given Intent specifications in order to generate ground ones that amount to executable Intent test cases. Thus the Intent specification language connects generators to executors of Intent test cases smoothly with a fuzzing strategy.

We design a fuzzing strategy that comes from the characteristics of how an Android component handles incoming Intents. Most of Android components start with a series of conditional statements branching according to the action name of an incoming Intent, as shown in Figure 1. Then most of instances of Intent vulnerability are likely to be either in the body of each conditional statement that runs when some of matching action is found, or the body of the "else" part that runs when no action is matched.

Our fuzzing strategy is to generate ground Intent specifications either compatible or incompatible to a given Intent specification. A ground Intent specification is compatible to an Intent specification if it has fields described by the given Intent specification optionally having more fields. A compatible ground Intent specification is interpreted as one of Intents that the given Intent specification denotes. For example, the former disjunct with EDIT action in the previous example of Intent specification can be refined into:

```
{ cmp = Activity com.example.android/.Note
  act = android.intent.action.EDIT
        [ RzUrx7    = boolean True ,
          HR7Ja6d7 = String AHMlyG0z3jjErO ]
  dat = URI qoFXwARtpfV-LNN }
```

where extra data with two random keys *RzUrx7* and *HR7Ja6d7* are introduced with one random boolean value and the other random string, and an extra Intent data field (*dat*) is also created with some malformed URI.

An incompatible ground Intent specifications is either a field-structure-preserving one or a random one. A field-structure preserving ground Intent specification shares the *skeleton* of a given Intent specification. For example, if the given Intent specification has an action, it has some action too, not necessarily the same action name described in the given specification. It can have new extra fields as well. For example, the former disjunct with EDIT action in the previous example of Intent specification can lead to:

```
{ cmp = Activity com.example.android/.Note
  act = android.intent.action.ADD
        [ key2 = int[] -1233387, -72316,
          dKQn = String xZQbcCTOW]
  typ = video/* }
```

where the action name becomes different, new extra data and type (*typ*) are introduced.

The other kind of incompatible ground Intent specification is a randomly generated one. As the name stands for, a random ground Intent specification can have arbitrary fields and values only preserving the *cmp* field which holds a target component name. For example,

```
{ cmp = Activity com.example.android/.Note
  dat = tel:123
  cat = [ ttoIjEWJnpk, vYQEpERvvb, xpWj_Q,
          android.intent.category.APP_CALENDAR] }
```

where the field of action with EDIT disappears. It has an Intent data (*dat*) with a telephone number and a category (*cat*) associated with a random array value.

Compatible ground Intent specifications will attempt to detect any inappropriate handling of malformed Intents inside the body of the conditional statement for each known action in an Android component. With incompatible ground Intent specifications, we anticipate an encounter with such an Intent vulnerability

inside the body of the "else" part or inside the body of the conditional statement for some unknown action. Later, we will see that our strategy is good enough to uncover many instances of Intent vulnerability in evaluation with real-world Android apps.

4.2. Executing Intent Test Cases via ADB Commands

One form of Intent executors in our tool is done via the ADB interface after we transform ground Intent specifications into ADB commands. Three ADB commands are transformed from one compatible and two incompatible ground Intent specifications explained previously, as follows:

```
adb shell am start -n com.example.android/.Note
  -a android.intent.action.EDIT
  --ez RxUrx7 True
  --es HR7Ja6d7 AHMlyG0z3jjErO
  -d qoFXwARtpfV-LNN

adb shell am start -n com.example.android/.Note
  -a android.intent.action.ADD
  --eia key2 -1233387, -72316
  --es dKQn "xZQbcCTOW"
  -t video/*

adb shell am start -n com.example.android/.Note
  -d tel:123
  -c ttoIjEWJnpk, vYQEpERvvb, xpWj_Q, android.intent.category.APP_CALENDAR
```

The transformation is straightforward. According to the syntax of ADB commands [1], the prefix "adb shell am start" directs Android platform to launch some Activity named by the option -n. The option -a is for action name, –ez is for extra boolean, –es is for extra string data, –eia is for integer array, -d is for URI to some data, -t is for the type of the data that the URI points to, and -c is for category. For Service and Broadcast Receiver components, the prefix starts with "adb shell am startservice" and "adb shell am broadcast", respectively, and the rest of options are given in the same manner.

Once a set of ADB commands are ready for each associated Android component in an Android app, we repeat the following procedure with each ADB command in sequence. Our tool clears any previously running instance of a target Android app, for example, by "adb shell am force-stop" with the package name of the application. It executes an ADB command to launch an Android component in a target app and to give the component an Intent designated in the command. It waits a period of time collecting Android logs flowing from running the Android app via *logcat* in the ADB interface. It analyzes the collected logs to decide if the Android component gets terminated abnormally or not by finding some textual patterns in the logs to be explained below. It writes all Android logs and the analysis result in a result file.

Note that the following table shows textual patterns that our tool uses to judge a failure from Android logs when Activity, Service, and Broadcast Receiver crash.

Component Type	Textual Patterns for Failures
Activity	ActivityManager: Force finishing activity
Service	AndroidRuntime: Shutting down VM
Broadcast Receiver	ActivityManager: Process pid **** has died.

Our tool is designed to write all Android logs and the analysis result in a Microsoft Excel file as shown in Figure 6. Using MS Excel format turns out to be very useful because it allows us to highlight occurrences of failures in red color and to review a very large amount of logs easily through MS Excel program without developing any viewer.

According to our experience with our tool, running too many Intent test cases by ADB commands is sometimes found to get the ADB interface unstable, and so one might not be able to get any Android logs

E	com.chbre	AndroidRu	Caused by: java.lang.NullPointerException: Attempt to invoke virtual method 'boolean java.lang.String.contains(java.lang.CharSequence)' on a null object reference
E	com.chbre	AndroidRu	at com.chbreeze.jikbang4a.support.WebViewSupport$WebViewClient.handleZigbangUrl (WebViewSupport.java:107)
E	com.chbre	AndroidRu	at com.chbreeze.jikbang4a.UriActivity.onCreate(UriActivity.java:17)
E	com.chbre	AndroidRu	at android.app.Activity.performCreate(Activity.java:6020)
E	com.chbre	AndroidRu	at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1105)
E	com.chbre	AndroidRu	at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2284)
E	com.chbre	AndroidRu	... 10 more
W		ActivityMa	Force finishing activity com.chbreeze.jikbang4a/.UriActivity

Figure 6: An Example of Failure Logs in Microsoft Excel

further from the Android app on testing. This will prevent us from running our tool over a large number of Android apps at the same time in practice. To overcome this difficulty, we design our ADB executor to take about five seconds in each run. This delay is required to test one Intent test case after another without any interference between the two subsequent tests, according to our experience. In addition, we build a simple watch dog program to monitor the status of the ADB interface and to reboot the Android mobile phone whenever the watch dog program sees some blocked state with no progress. Some detail on this will be explained later in discussion. Thanks to this auxiliary watch dog program, we have successfully finished our experiment with 50 real-world Android apps in a batch run of our tool without stopping in the middle.

4.3. Automatically Reporting Failures Without Duplication

A naive way of reporting failures found in Android logs by the ADB command executor turns out to be not so effective because it can report many occurrences of the same failure happening in the same program point and throwing the same runtime exception. According to our experiment to be shown later, 77% of the total number of failures have appeared again, and so the ratio of unique failures amounts to only about 23%. With many Intent test cases on an Android app or with many Android apps to test, the fuzz testing tends to produce many same failures demanding too much efforts in classifying different failures. Hence, it is required to classify failures into sets of the similar failures for reduction of such efforts in the post-classification.

Our tool employs a procedure of making groups of similar failure logs shown in Algorithm 1. The input is a list of failure logs obtained from the fuzz testing on an Android component, and the output is a set of index sets, each index set of which tells occurrences of each failure.

Input: List : a list of N failure logs
Output: G : a set of index sets to the failure logs
$G = \{\{1\}, \cdots, \{N\}\}$
for $failure\ log_i$, $failure\ log_j \in List$ **do**
  $lcs = ComputeLCS(failure\ log_i, failure\ log_j)$
  $similarity = |lcs|\ /\ max(|failure\ log_i|, |failure\ log_j|)$
  if $similarity \geq 0.99$ **then**
    Merge G_i and G_j in G such that $i \in G_i$ and $j \in G_j$
  end
end

Algorithm 1: Classifying Failure Logs

The procedure is based on the longest common subsequence (LCS) algorithm *ComputeLCS* [8]. This traditional algorithm discovers the LCS, which is not necessarily contiguous, from given two sequences. For example, given two input sequences ABCBDAB and BDCABA, it finds BCA and BDA for length three LCS, BCBA and BDAB for length four LCS, and nothing for length five LCS. Therefore, it eventually outputs the two sequences of length-four LCS.

Note that it is straightforward to get a character sequence by concatenating rows in a failure log, such as one shown in Figure 6, for the LCS algorithm will take as input.

Android Apps	Version	Bytes in Dex	# of Dex Files
between	2.6.1	9,277,552	2
Camera360	6.1.2	7,759,816	1
cleanmaster	5.9.7	9,325,319	2
excel	16.1.0.1	9,308	1
facebook	31.0.0.20.13	1,692,552	1
gsshop	2015.5.7	4,327,352	1
HoHo	2.1.96	10,523,224	2
instagram	6.21.2	4,756,136	1
kakao story	3.0.3	6,972,372	1
kakao talk	4.8.2	6,682,188	1
line	1.0.0	10,270,968	2
lovers	1.6	3,182,240	1
melon	3.2.1	6,500,948	1
moonreader	3.0.6	5,668,968	1
MarbleGame	1.9.37	4,230,464	1
UriBankOneTouch	1.1.0	2,407,880	1
YonseiWonjuApp	0.1	1,400,052	1
JikBang	4.9.27	10,529,324	2
NaverBooks	2.1.13	8,563,220	2
DaumMap	3.9.8	6,751,636	1
AlYak	1.6.1.2	5,051,092	1
TexView	3.9.7	859,104	1
KBStarBanking	4.0.8	4,645,264	1
GiniMusic	4.00.00	12,147,888	2
rolling sky	1.2.3	8,358,132	1
LotteCinema	2.31	7,277,628	1
LotteHomeShopping	1.5.8	7,034,824	1
CacaoNavi	3.4.0	6,490,984	1
Himart	3.05	10,551,412	2
AppExtractor	1.3.5	1,136,644	1
ediyaMembers	1.0.7	6,150,464	1
Twitter	6.12.0	18,413,280	3
Myjio	3.2.10	12,192,348	2
Free International Calls	2.2.12	1,985,000	1
360 Security Antivirus Boost	3.7.5	10,736,504	2
Text Editor	1.2.b13	5,494,668	1
Cloud VPN Free Unlimited	1.0.4.4	5,114,280	1
MemoRemember	1.1.5	3,713,720	1
CGV	4.2.0	6,217,276	1
HanaMembers	1.0.44	7,637,936	2
CacaoMusic	4.1.7	6,529,852	1
4Shared	3.35.0	23,672,136	3
Band	5.6.0.1	18,871,340	3
Turbo Cleaner	1.0.0	5,262,600	1
Naver	6.0.5	6,786,036	1
IndepCapaign	2.0.5	7,773,000	1
MobileTMoney	6330G	9,183,752	1
Auction	4.5.70	6,765,616	1
Coocha	4.14	5,967,572	1
HappyPoint	5.1.1	14,877,340	2
Average	n/a	7,354,584	1.4

Table 1: Fifty Android Apps from Marketplace for Evaluation

Our algorithm initially lets each failure log have an index set to itself. For each combination of two failure logs indexed by i and j in the input list, the algorithm computes the longest common subsequence lcs, and then it computes a similarity between the two failure logs by the ratio of lcs to the longer failure log. Whenever a similarity number is high, the algorithm regards the two failure logs as the same one, merging the two index sets to the failure logs. We use 99% as the level of similarity to judge that the two logs hold the same failure, merging the associated index sets into one on the high similarity. The reason that the highest level (100%) is not used is that we want to allow some minor differences such as thread IDs that are numbered differently per each run.

Having many duplicate failures in the existing Intent fuzz testing tools is thought to be common, and the proposed algorithm can also be applied to the tools to reduce much efforts on classifying duplicate failures manually.

5. Evaluation

We present an experimental result on applying our Intent fuzz testing tool to 50 popular Android apps downloaded from Google Play where no source code is provided. The Android app names, versions and DEX binary code sizes are listed at Table 1. The source code of our tool, all Android APK files used in the experiment, and the result data are available in our companion web site [9].

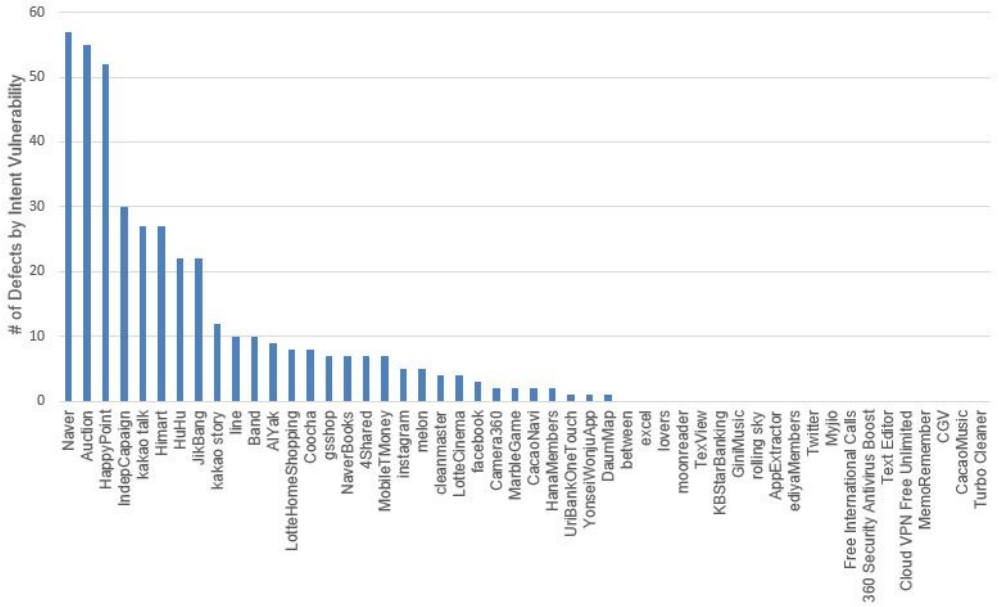

Figure 7: The Number of Discovered Failures Due to Intent Vulnerability

5.1. Failure Counts Due to Intent Vulnerability

First, the experiment result supports that the problem of Intent vulnerability on Android apps is serious: the two third of Android apps on the experiment have experienced abnormal crashes due to Intent vulnerability. Our Intent fuzz testing tool has discovered total 409 different failures due to Intent vulnerability over 50 Android apps as shown in Figure 7. It amounts to 8 failures per Android app on average. All the reported failures caused the associated Android apps to crash during the Intent testing.

Second, the ratio of failure counts over binary code sizes is a simple and useful criterion to classify robust and weak Android apps in terms of Intent vulnerability. Figure 8 shows the analysis result: the increasing rate of the number of failures is roughly 70% of the increasing rate of the sizes of Android apps according to the simple linear regression analysis. We measured the size of Android apps in DEX binaries. Based on the ratio of failure counts over Android binary code sizes shown in Figure 8, the seven Android apps above the dotted linear line are evaluated as weak ones. They are actually the top seven Android apps in the number of discovered failures as shown in Figure 7. Interestingly, they are all developed by the domestic companies, and it remains to see if there is any good reason for the weakness of domestic Android apps. The three big Android apps located below the dotted linear line are considered as very robust ones: 4shared (7 failures/23.67MB), Band (10 failures/18.87MB), and Twitter (no failures/18.41MB).

5.2. Automatic Classification of Failures in Android Logs

Figure 9 shows how effective our tool using LCS algorithm is in classifying duplicate failures from Android logs by the ratio of the number of failure groups to that of all failures. It has found 79.2% of all failures in a single Android app reappear on average, and so it has successfully excluded much efforts on further examination. For example, HanaMembers shows the best performance by merging 71 failures into only two groups of duplicate failures, and we have only to examine 2.8% of all the failures for further analysis.

Figure 9 enumerates only thirty Android apps where the tool discovered at least one failure of Intent vulnerability. For the twenty four Android apps (80%) from 4Shared to HanaMembers, our tool removes duplicate failures more than a half of all (with the ratio below 50%) where we clearly see the effectiveness of the proposed automatic classification method.

When we look at the rest six Android apps (20%) with the ratio above 50%, the first four Android apps Facebook (3 failures with 3 groups), YonseWonjuApp (1 failures with 1 group), DaumMap (1 failures with

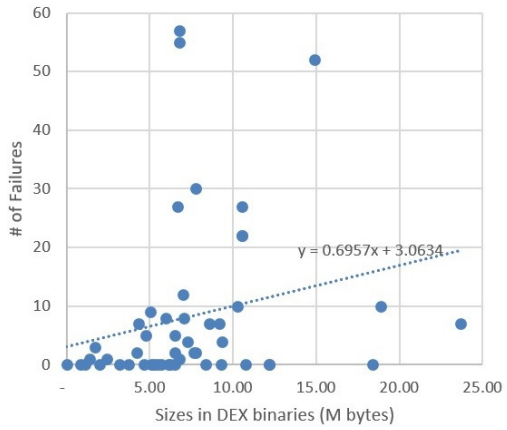

Figure 8: Number of Failures vs. Binary Sizes

1 group), CleanMaster (5 failures with 4 groups), and the sixth Android app, Instagram (8 failures with 5 groups) have failure counts below the average so that the high ratio of the number of failure groups to that of all failures does not make sense much.

Only the fifth Android app, Auction, with 76 failures identified into twelve groups worth a further analysis. The grouping result is $G_0 = \{0, 1, 2, 4, 5, 6, 8, 11, 12, 13, 18, 20\}$, $G_1 = \{7, 9, 10, 14, 15, 16, 17, 19, 21, 22\}$, $G_2 = \{23\}$, $G_3 = \{24\}$, ..., $G_{54} = \{75\}$. The Android log #0 consists of 41 lines and about 9 KBytes characters while the log #23 and the log #24 consists of 688 lines and about 140 KBytes characters and 430 lines and about 100 KBytes characters, respectively. The logs from #23 to #75 in Auction are extraordinarily large when the sizes are compared with the size of logs from the other Android apps. For example, the Android log #0 of Facebook consists of 67 lines and 13 KBytes characters. Although in the case of the Android logs #23 and #24 of Auction, we confirm that they are different from each other by inspection, it could be more likely to miss detecting duplicates as Android logs are large. In this respect, the LCS-based failure classification method can be improved more because it is not able to detect duplicates when the same lines in the duplicates appear in different orders. It would be interesting to see if we could employ more discerning method such as machine learning algorithms.

5.3. Execution Time for Intent Fuzz Testing

The evaluation of our tool in terms of the execution time for Intent fuzz testing leads to two observations, which have been rarely discussed before. First, the execution time for Intent fuzz testing on a single Android app is proportional to the number of Intent test cases (i.e., ADB commands) used for the testing, and it is not affected so much by the time for classifying failures using LCS algorithm in general, which will be justified by the following results.

Figure 10 shows the execution time for Intent fuzz testing, which is the time for running Intent test cases plus the time for grouping. Testing all fifty Android apps took 570,255 (545,617+24,637) seconds for running Intent test cases and grouping. On average, it took 3.17 hours for testing a single Android app. Only 14 Android apps took less than an hour, and the rest took more than that.

The time for Intent fuzz testing on a single Android app is found to be proportional to the number of Intent test cases. Table 2 tells us that for the fifty Android apps, our tool automatically constructs 3,408 Intent specifications (68.2 per an Android app on average) by the method explained in Section 4.1.1. The Intent specifications expand to 102,240 Intent test cases (ground Intent specifications) by generating 30 Intent test cases per each Intent specification. We have tested 2,045 Intent test cases over a single Android app on average, and each Intent test case requires about 5 seconds due to the reason explained in Section 4.2. Based on the figures, we can estimate the time for testing a single Android app, which is 2.84 hours (=2,045 x 5 / 3,600), which is quite close to the time (3.17 hours) obtained from the experiment.

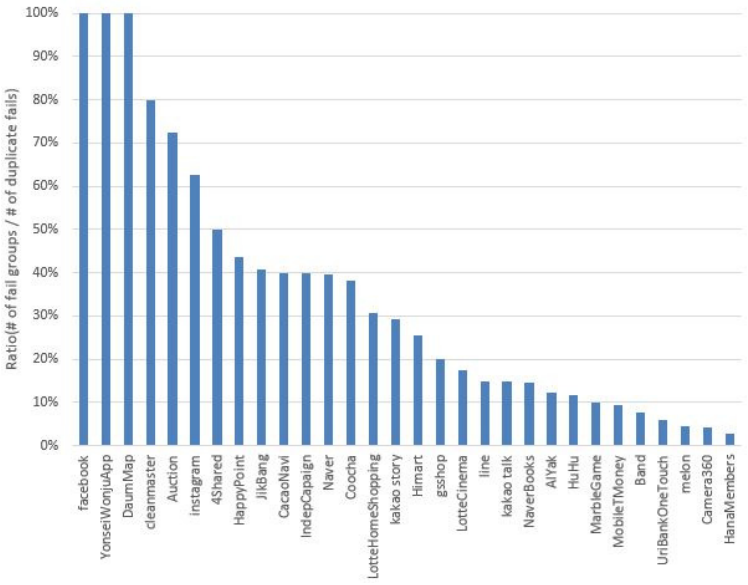

Figure 9: Ratio of the number of failure groups to that of all failures

The time for grouping similar failures is small when compared with the time took for the pure testing time. It took 4.3% of the time for running Intent test cases on average. However, it sometimes happened that the time for grouping grows extraordinarily. According to Figure 10, Auction took more time for grouping failures than that for running Intent test cases, and HappyPoint took longer time for grouping failures than the other Android apps (except Auction) do. The reason is that the length of the failure logs are relatively long and the number of the failure log groups is large. In such an exceptional case, the time for classifying failures does not depend on the number of Intent test cases any more.

Second, our Intent fuzz testing tool itself is robust to run fully automatically, which is important in practice particularly when it is applied to many Android apps by batch processing. This allowed us to evaluate the total execution time for Intent fuzz testing on all fifty Android apps, which has never been reported before. For the fifty Android apps, our tool ran 7.1 days without human intervention until it finished Intent fuzz testing on them. For this to work successfully, we employed a monitoring mechanism of the tool that detected a blocked state of running an Android app and rebooted the mobile phone to restart testing the app. Testing 14 of the 50 Android apps have experienced such a reboot of the mobile phone. A finding on the unstable status of Android apps during fuzz testing was reported in other research [3] as well, but no countermeasure was discussed except human intervention.

Based on the measured testing time by our Intent fuzz testing tool, it would be interesting to attempt to estimate how long it will take to test with our tool all Android apps in Google Play. It is known that Google Play provides 2.81M Android apps as of December 2016. The average time for Intent fuzz testing in our experiment was 2.84 hours per a single Android app. When we estimate the Intent fuzz testing time for all Android apps in Google Play, it takes 332,516.7 days.

In summary, this testing time analysis suggests that some consideration should be taken for efficiency of Intent fuzz testing. First, we could configure our tool to test multiple Android apps in parallel, or we could test a single Android app installed in multiple mobile phones simultaneously for speed up. Second, we could start with more precise Intent specification to generate the less number of Intent test cases. Some static and dynamic analyses discovering the structure of Intents of interest would be helpful for the purpose.

5.4. Root Cause Analysis on Intent Vulnerability

This section reports a root cause analysis on failures due to Intent vulnerability found by our experiment. Despite the nature of randomness in Intent fuzz testing, our experiment has discovered an interesting failure

Android Apps	# of Intent Specs	# of Intents	# of Failures	# of Groups
between	78	2340	0	0
Camera360	60	1800	49	2
cleanmaster	80	2400	5	4
excel	14	420	0	0
facebook	288	8640	3	3
gsshop	56	1680	35	7
HoHo	148	4440	191	22
instagram	38	1140	8	5
kakao story	86	2580	41	12
kakao talk	182	5460	183	27
line	68	2040	67	10
lovers	10	300	0	0
melon	120	3600	114	5
moonreader	12	360	0	0
MarbleGame	16	480	20	2
UriBankOneTouch	22	660	17	1
YonseiWonjuApp	14	420	1	1
JikBang	28	840	54	22
NaverBooks	34	1020	48	7
DaumMap	50	1500	1	1
AlYak	54	1620	73	9
TexView	6	180	0	0
KBStarBanking	30	900	0	0
GiniMusic	112	3360	0	0
rolling sky	20	600	0	0
LotteCinema	24	720	23	4
LotteHomeShopping	48	1440	26	8
CacaoNavi	10	300	5	2
Himart	44	1320	106	27
AppExtractor	2	60	0	0
ediyaMembers	10	300	0	0
Twitter	98	2940	0	0
Myjio	84	2520	0	0
Free International Calls	50	1500	0	0
360 Security Antivirus Boost	128	3840	0	0
Text Editor	10	300	0	0
Cloud VPN Free Unlimited	8	240	0	0
MemoRemember	14	420	0	0
CGV	244	7320	0	0
HanaMembers	42	1260	71	2
CacaoMusic	34	1020	0	0
4Shared	62	1860	14	7
Band	86	2580	132	10
Turbo Cleaner	30	900	0	0
Naver	224	6720	144	57
IndepCapaign	22	660	75	30
MobileTMoney	196	5880	75	7
Auction	40	1200	76	55
Coocha	192	5760	21	8
HappyPoint	80	2400	119	52
Total	3408	102240	1797	409

Table 2: An Experiment Result for Intent Vulnerability

that the other researches have not reported before.

Table 3 shows a summary of statistics on kinds of exceptions due to Intent vulnerability discovered by our Intent fuzz testing tool on the fifity Android app.

The most frequent root cause of Intent vulnerability is the Null pointer reference exception. A close examination on the failure logs with this exception tells that some missing field in an Intent test case caused raising the exception. For example, in the following line excerpted from a failure log on Auction, an Uri expected for the data field of an Intent test case is missing, resulting in the exception.

- Caused by: java.lang.NullPointerException:
  - Attempt to invoke virtual method 'java.lang.String android.net.Uri.getScheme()'
  - on a null object reference

The ClassNotFound exception, which is found only on Naver, reveals a serious mismatch between the package name of a class and its declaration on AndroidManifest.xml. One of the associated failure logs is as follow:

- Caused by: java.lang.ClassNotFoundException:
  - Didn't find class "com.nhn.android.search.ui.picturerecognition.BarcodeRecognitionActivity" ...

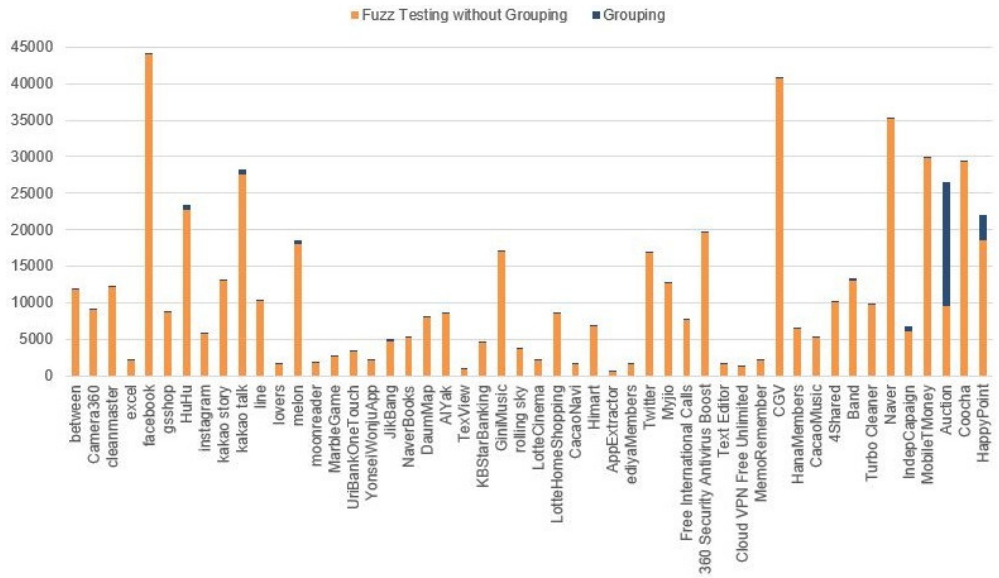

Figure 10: Time for Intent Fuzz Testing

where the mentioned class BarcodeRecognitionActivity is actually found in differently named package com.nhn.android.search.ui.recognition, according to a reverse engineering analysis on the Naver Anroid app. But the wrong package name is declared in the AndroidManifest.xml of the application so that it is legal to attempt to invoke BarcodeRecognitionActivity with this wrong package name. This is an example of invalid configuration of an Android app. Such a misconfiguration does not cause any problem in compile-time, but it can cause a runtime exception.

Many of the NumberFormat exceptions, which are also found only on Naver, have been caused by some extra field set with values of types different from integer. To prevent this exception, validity of values from Intents should be considered carefully before the values are retrieved from Intents.

- Caused by: java.lang.NumberFormatException: Invalid int: "UijJLfrRYfATQmd"
  - at java.lang.Integer.invalidInt(Integer.java:138)
  - at java.lang.Integer.parse(Integer.java:410)

The UnsupportedOperation exception is seen on those Intent test cases with some invalid Uri. When it comes to an excerpt from the following failure log, we tested with a Uri "tel:xxx" where there is no query part. Values such as a Uri have some structure which makes the validity check more difficult than primitive values such as Integer.

- Caused by: java.lang.UnsupportedOperationException: This isn't a hierarchical URI.
  - at android.net.Uri.getQueryParameter(Uri.java:1665)

Although it is rare, IllegalArgument exception is found to be thrown. The associated method is accessible because it is declared as public but intends to be internal according to the documentation in Android source code by Google. Without further information on the relevant Android apps, it is not easy to figure out the execution path leading to an invocation of such an internal class API.

- Caused by: java.lang.IllegalArgumentException:
  - Expected com.google.inject.internal.util.FinalizableReference.

Android Apps	Null Pointer	Unknown Exn	Illegal Argument	Unsupported Operation	Class NotFound	Number Format
Camera360	0	2	0	0	0	0
cleanmaster	4	0	0	0	0	0
facebook	0	3	0	0	0	0
gsshop	5	0	2	0	0	0
HoHo	21	1	0	0	0	0
instagram	5	0	0	0	0	0
kakao story	12	0	0	0	0	0
kakao talk	22	0	0	5	0	0
line	10	0	0	0	0	0
melon	0	0	0	5	0	0
MarbleGame	0	0	0	2	0	0
UriBankOneTouch	0	0	0	1	0	0
YonseiWonjuApp	0	1	0	0	0	0
JikBang	20	2	0	0	0	0
NaverBooks	7	0	0	0	0	0
DaumMap	0	1	0	0	0	0
AlYak	5	3	0	1	0	0
LotteCinema	4	0	0	0	0	0
LotteHomeShopping	8	0	0	0	0	0
CacaoNavi	0	2	0	0	0	0
Himart	27	0	0	0	0	0
HanaMembers	2	0	0	0	0	0
4Shared	7	0	0	0	0	0
Band	9	1	0	0	0	0
Naver	1	2	0	0	30	24
IndepCapaign	30	0	0	0	0	0
MobileTMoney	3	4	0	0	0	0
Auction	55	0	0	0	0	0
Coocha	0	0	0	8	0	0
HappyPoint	8	44	0	0	0	0
The rest	0	0	0	0	0	0
Total	249	66	2	22	30	24

Table 3: A Statistics on Exceptions Causing Intent Vulnerability

– at com.google.inject.internal.util.$Finalizer.startFinalizer(Finalizer.java:77)

For the second entry in the Table 3, the relevant failure logs show few clue for us to decipher the causes of Intent vulnerability partly because there is no source available for Android binary apps for the experiment.

6. Related Work

6.1. Intent Fuzzing for Testing Robustness of Android Apps

Recent researches [2, 3, 4, 5, 6, 7] have developed *Intent fuzzer*s to detect Intent vulnerability. Their Intent fuzzers generate arbitrary (possibly malformed) Intents using their own strategies, and test with the Intents if running of each Android app crashes. It has been reported that the Intent fuzzers have uncovered many interesting instances of Intent vulnerability in real-world Android apps.

Null IntentFuzzer [2] was the first tool for testing robustness of Android apps. It has the form of an Android app. It gathers information on installed applications and their Intent filters through Android API. It sets NULL for all fields of Intents to test Android apps with.

Maji et al [3] extended the first Intent fuzzer to develop a new one called JarJarBinks (JJB), which is a standalone Android app capable of retrieving a list of installed Android apps together with Intent filters. It generates both valid and semi-valid Intents based on the retrieved information under four strategies: semi-valid action and data, blank action or data, random action or data, random extra data.

With JJB, they measured the number of failed components for various types of components. For instance, 29 (8.7%) out of total 332 Activities crash with generated semi-valid Intents on Android 4.0 emulator. The distribution of exception types is also measured to understand how components fail due to uncaught exceptions. It is shown that `NullPointerException` makes up the largest share of all the exceptions. In case of implicit Intents, the number of crashes due to `NullPointerException` is 32 (38.5%) out of the total 83 crashes. Other exceptions like `ClassNotFoundException` and `IllegalArgumentException` are next significant ones.

It is reported that JJB is a semi-manual approach. First, when a system alert was generated due to application crash, a user is involved in closing the alert dialog boxes. Second, when an Activity was started

as a new task, JJB could not close it in an automatic manner easily since it is an Android app. Third, JJB requires a user's intervention to stop, for example, a thread hang [3].

DroidFuzzer [4] focused on the data field of Intents being set with malformed audio and video files only for Activity type components. Based on the extracted URI and MIME data type information from the AndroidManifest.xml file, it built pieces of abnormal audio and video data for testing a target Activity. The tool is equipped with a dynamic crash monitoring module that is capable of detecting Activity crashes and native code crashes. DroidFuzzer uncovered bugs such as consumption of resources, ANR (Android Not Responding), and crashes from not dealing with malformed audio and video files well, rather than bugs resulting from Intent field missing or incorrect types of Intent field values.

IntentFuzzer [5] combined a static analysis with random fuzzing to dynamically test Android apps. A path-insensitive, inter-procedural CFG analysis was employed to extract the structure of Intents that a target component expects. The analysis involves traversing Dalvik bytecode instructions collecting all calls to Intent's getter/setter methods including calls to their bundle objects, starting from each component's entry point (e.g., onCreate method for Activity). The majority of the calls use a specific string key to extract extra data from Intents whereas data type itself is encoded in the name of the methods. This research also attempted to use Flowdroid [26] for static analysis on more precise Intent structure, but it reported that the simple CFG-based analysis mentioned previously is enough for Intent fuzz testing in terms of scalability and precision [5]. A set of Intents was generated afterward with the statically analyzed Intent structure information to explore more execution paths. Target components are executed with these fuzzed Intents, and both code coverage and crashes due to exceptions were monitored. This research contrasts to the two previous researches [3, 4] where static analysis in the two researches mean the extraction of Intent structure information from the Android manifest information file.

IntentDroid [6] addressed eight kinds of vulnerabilities in Activity (and Fragment) due to Intent component communication including Java crash, Fragment injection, UI spoofing, and Cross-application scripting (XAS). The purpose of using this tool is not only to detect Java crash as JJB, DroidFuzzer, and IntentFuzzer have aimed at, but it is also to discover other kinds of vulnerable Android apps such as those exploiting unsafe dynamic Fragment loading process and those injecting JavaScript code into HTML-based UI to access sensitive information and to spoof UI to trigger phishing attacks. It featured high coverage with low overhead by monitoring some selected set of Android platform APIs responsible for security-relevant functionality as well as access to Intent fields and by utilizing the monitored information to guide testing.

ICCFuzzer [7] is another interesting tool to uncover crashes by Null reference exception, Intent spoofing, Intent hijacking, and data leak by path-insensitive interprocedural CFG static analysis and proper Intent and event generation using the analyzed information on Intent structure, string constants, and events relevant to the ICC vulnerabilities. It was applied to Android apps from DroidBench [30] and Google Play comparing the number of detected vulnerabilities with that by IntentFuzzer [5] and Null IntentFuzzer [2].

6.2. Verifying Intent for Security and Privacy Leak Detection

Although the scope of this paper is Intent vulnerability associated with the robustness of Android apps, there have been several researches on Intent vulnerability associated with security and privacy issues such as personal data loss and corruption, phishing, and other unexpected bad behavior. ComDroid [19] is a tool to check if a given Android app is exploitable when servicing external Intent data, making use of meta data declared in AndroidManfiest.xml together with specific API usages.

CHEX [20] performed an in-depth program analysis to check Android apps for component hijacking vulnerabilities. This tool has been applied to an extensive set of Android apps to uncover the potential vulnerabilities, measuring execution time for the analysis.

Avancini and Ceccato [21] proposed a test case generation strategy and a testing adequacy criterion for Android apps, checking if any Intent violates the Intent filter, if Android apps fail to validate Intents and do not reject it, and if Android apps execute a protected API call under a special permission while performing an action requested by Intent.

AppsPlayground [22] is a modular framework for dynamic analysis of Android apps to detect malware (which has a malicious Intent) and grayware (which are not malicious but, for example, may leak private

information for a legitimate purpose without user's awareness). The framework offers multiple detection techniques such as event triggering and context-sensitive execution.

Kun Yang et al's IntentFuzzer [23] showed a dynamic Intent fuzzing mechanism to uncover violations of permission model for Activity and Intent-started Service typed components. This violation is due to capability leak (permission re-delegation) where a benign Android app with legitimate permissions performs actions on request by unprivileged Android apps sending Intent to the benign one without checking if the request Android apps possess the privilege or not.

APSET [24] designed a model-based testing method based on ioSTS formalism for the detection of Intent-based vulnerabilities in Android apps. Under the formalism, vulnerability patterns can be specified.

Ghio et al [25] also dealt with a particularly dangerous case of the permission re-delegation problem, called Android Wicked Delegation (AWiDe). The vulnerability consists in executing privileged tasks on behalf of another Android app but with additional preconditions that the privileged tasks are executed with data coming from attacking Android apps, controlled by the attacker and without performing adequate validation of such data.

6.3. Other Fuzzing-based Testing for Android Apps

The fuzzing-based technique has been applied to testing other aspects of Android apps: testing on Android system service APIs [10] by generating random arguments on the APIs, testing Android low-level IPC (Inter-process communication) [11] by designing an automatic testing framework supporting parameter-aware fuzzing called BinderCracker, testing Android GUI with generated text input on UI elements in a comparative study of the existing test input generation techniques for Android [12], detecting Android apps with poor input validation by CrashFuzzer [13], and testing memory leak in Android apps [14].

There have been some notable researches on automatic Android GUI testing tools and techniques: A2T2 [15] and AndroidRipper [16]. The tools have dynamically analyzed Android apps to get a list of replayable events in GUI widgets, and have generated sequences of graphical events and sensor events. Some code was instrumented to record crashes and to eventually translate event sequences into JUnit test cases. N. Mirzaei used symbolic execution and combinatorial input generation techniques for Android app testing [17]. CRASHSCOPE [18] is an automatic tool to explore a given Android app using systematic input generation according to several strategies informed by static and dynamic analyses. Whenever a crash was detected, this tool generated an augmented crash report containing screen shots, detailed crash reproduction steps, the captured exception stack trace, and a fully replay-able script that automatically reproduces the crash on a target device.

7. Discussion

7.1. Comparison with Existing Intent Fuzzing Tools for Detecting Crashes

We compare our Intent fuzzing tool, named *Hwacha*, with the existing tools for detecting crashes due to Intent vulnerability [2, 3, 4, 5, 6, 7] in terms of generating Intent test cases, executing testing, and post processing of testing results for review. First, an Intent fuzzing tool had better have a flexible way of Intent test case generation to take information on Intent structure from various sources.

	Empty /Random	Android Manifest.xml	Static Analysis	Dynamic Analysis	Generic Provision
Null IntentFuzzer [2]	Y				
JJB [3]	Y	Y			
DroidFuzzer [4]	Y	Y*			
IntentFuzzer [5]	Y	Y	Y		
IntentDroid [6]	Y	Y		Y	
ICCFuzzer [7]	Y	Y	Y†		
Hwacha (this paper)	Y	Y	‡	‡	Y

This table explains that all Intent fuzzing tools are based on random generation for Intent test cases. Except Null IntentFuzzer [2], all the tools make use of information from Intent Filter declared in AndroidManifest.xml for Activity, Service, and Broadcast Receiver. DroidFuzzer [4] confines itself to testing Activity using Uri data pointing to fuzzed audio and video contents, which is different when it is compared with JJB [3] focusing more on missing actions, malformed extra data and so on. A mark * in the table tells this difference. Both IntentFuzzer [5] and ICCFuzzer [7] have used FlowDroid [26] for backend of their own static analyzer, for example, to collect keys and types of extra data of Intent statically. ICCFuzzer [7] went one step more to collect event handler information in Activity and Service components potentially to trigger deeper execution of target Android components, which a mark † points out. IntentDroid [6] is the only tool to be based on dynamic analysis to collect information on Intent structure such as keys and types of extra data of Intent.

Hwacha offers a flexible mechanism to get Intent structure information written in the Intent specification language. It is true that the Intent structure information obtained static and dynamic analyses in IntentFuzzer [5], IntentDroid [6], and ICCFuzzer [7] can be straightforwardly written to Intent specifications. In addition, Hwacha provides SQA teams or programmers an easy way to test the apps with specific Intent test cases manually constructed based on their domain knowledge. To point out this reason, we have a mark ‡ in the static and dynamic analysis columns, and we have Y in Generic Provision column of Hwacha in the table. Currently, Hwacha is the only Intent fuzzing tool providing this capability.

Second, an Intent fuzzing tool should be fully automatic in executing testing particularly when it is applied to batch processing over many Android apps. There are a few considerations on this automatic procedure. Basically, it had better a facility for automatic installation and uninstallation of each Android app before and after executing testing. During the execution of testing, it is also required to have a mechanism to detect crashes during running an Android app automatically, for example, as is explained in Section 4.2. DroidFuzzer [4], IntentDroid [6], and ICCFuzzer [7] have mentioned how crashes are detected automatically, though there have not been so much details. In addition, the execution of testing should be performed reliably. We have experienced that Android apps often get stuck when too many Intent test cases are executed for testing. Hwacha is equipped with a kind of watch dog to monitor the running of Android apps by measuring the running time. If an Android app runs too long, the watch dog mechanism kills it and runs it again. It repeats 10 times, and if it still has some problem, it reboots the Android phone. According to our experience, a single rebooting is enough to make progress in case of having some trouble, resolving the problem.

The presence of a fully automatic tool immediately allows one to measure execution time for Intent fuzz testing, which is important in practice. The execution time has been rarely reported before. The only research work on IntentDroid presented very rough numbers as: it took three weeks to finish their testing over 80 Android apps, and it took several hours for each Android app on average [6].

Third, it is interesting to find that there is no previous work on attempting to removing duplicate failures in Intent fuzz testing. Due to the nature of fuzz testing, a tool can generate many different Intent test cases but resulting the same failure. More duplicate failures we have, more efforts we should make in reviewing the failures. Adopting LCS algorithm is a simple but very effective way to reduce the reviewing efforts. We believe that the LCS-based algorithm of removing duplicate failures can be generally usable in the existing Intent fuzzing tools: all the existing tools produce Android logs in the testing, which can be used as input of the algorithm.

The design that Hwacha outputs all Android logs in Microsoft Excel format is also very useful for review. The size of the output file may increase due to the use of this format when compared with that in a plain text. However, it was very helpful for us to examine the details of Android logs, and to process it by Java program since Java library for reading and writing Microsoft Excel documents is available. The existing research works have not emphasized this aspect. For reader's interest, all excel files in our experiment are available in a companion web site [9].

Hwacha has a capability to generate JUnit test code based on Android testing framework, which is useful for programmers with source code. When an Intent fuzzer tool is applied with Android source program, the automatic generation of JUnit test code will be helpful for building regression test suite on defects of Intent vulnerability discovered in the source program. Running the generated JUnit test code is via Android

Studio. The detail on this capability is explained in our companion web site [9].

7.2. How to Defend Intent Vulnerabilities Causing Crashes

The Intent vulnerability of causing Android app crashes is mainly due to the weakness of Android ICC design where there are few stringent mechanisms to enforce a contract between a sender and a receiver of an Intent. The mutual agreement on the well-formed Intent object is not thoroughly checked either statically in compile-time by Java compiler, as we discussed in Section 2, nor dynamically in run-time by Android platform.

Maji et al [3] have pointed out this matter to suggest two strategies to overcome the weak Android ICC design. First, one way to make Intent message format more explicit and therefore possible to capture is to use subclasses for Intent instead of a single flat type, for example, as:

```
public class CallIntent extends Intent {
  final static String action = "android.intent.action.ACTION_DIAL";
  Uri data;
  ComponentName cmp;
  ...
}
```

for the Intent class dedicated to activating Dial Activity. Then the Java compiler can now do automatic type checking since the messages use a type schema that the compiler is able to understand and enforce. However, this approach still has Intent vulnerability by not being able to enforce constraints on the values of data; the Java type checking does not avoid the Intent vulnerability from having Null in the Uri data, for example. Second, another way is to use a domain specific language to express the schema of various Intents, similarly as those approaches taken with many RPC systems which used an interface definition language (IDL). This IDL describes exactly the format of a remote invocation in enough detail so that the stub and skeleton code can be synthesized from this description.

Dart & Henson [27] is a library for Android which uses annotation processing to generate code that does direct field reading and assignment of extras of Intents. For example,

```
public class DealDetailModel {
   @InjectExtra String dealId;
   @InjectExtra @Nullable Boolean shouldShowMap;
}

// Intent consumption
@HensonNavigable(model = DealDetailModel.class)
public class DealDetailActivity extends Activity {
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     Dart.inject(detailModelInstance, this);
     ...
   }
}

// Intent creation
public class MainActivity extends Activity {
  ...
  Intent intent = Henson.with(this);
                  .gotoDealDetailActivity()
                  .dealId(dealId)
                  .shouldShowMap(true)
                  .build();
```

```
    startActivity(intent);
    ...
  }
```

where MainActivity activates DealDetailActivity with Intent described by DealDetailModel. The annotation @HensonNavigable is processed to generate an Intent builder with gotoDealDetailActivity, dealId, shouldShowMap, and build methods following the model described by DealDetailModel class. The caller Activity uses these methods to build an Intent to activate DealDetailActivity, which uses Dart.inject method to extract a string and a possibly nullable boolean value into dealId and shouldShowMap fields in a declarative way.

Jackson [28] is a suite of data-processing tools for Java to serialize passed data into a single Intent field and using it to check validity when unmarshalling. It has been developed for XML representation but can be applied to Android Intents.

The second author of this paper designed a runtime assertion library using Intent specification to defend incoming malformed Intents and to specify how to handle them, in his master's thesis [29].

```
  public class Note extends Activity {
  @IntentSpec(
    spec="{ act=android.intent.action.EDIT [ title=String, content=String ] }
       || { act=android.intent.action.INSERT }",
    exception={
      @IntentSpecException(
        error_code="EXTRA_FIELD_MISSING",
        process= // Code to initialize the Intent with default extras
        ),
      @IntentSpecException(
        error_code="default",
        process= // Code to finish this activity
        )})

    void onCreate(Bundle savedInstanceState) {
       // the same code as the body of the onCreate method in Figure 1
    }
    ...
  }
```

This example of the runtime assertion library starts from the example of Note Activity in Figure 1. The annotation @IntentSpec is processed to insert an automatically generated assert statement on incoming Intents in the beginning of the following method onCreate. The assert statement verifies Intents if they comply to the specified Intent specification. When they are not compliant to the specification, the assert statement executes one of the specified exception handlers chosen by error codes resulting from matching the Intent specification against malformed Intents, such as EXTRA_FIELD_MISSING. This shows another usage of the Intent specification language for verifying Intents in runtime.

8. Conclusion

We have developed a fully automatic Intent fuzz testing tool with two new features: a flexible structure in combining generators of Intent test cases with arbitrary executors and a mechanical method of failure classification. In our evaluation with 50 commercial Android apps, our tool uncovered more than 400 unique failures, including what have never been reported before, caused by Intent vulnerability on the Android apps. The automatic tally method excluded almost 80% of duplicate failures in all the crash logs, reducing efforts of testers very much in review of failures.

Our Intent fuzzer will be useful for Android app developers who are not much familiar with Android ICC design, for SQA team to test Android apps repeatedly, and for marketplace managers to examine a large number of Android apps.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2017R1A2B4005138). The authors thank to Seungwhui Lee, Hyeonsoon Kim, Sungbin Youn, and Jisun Choi for their efforts on our Intent fuzzer tool, Hwacha.

References

[1] http://developers.android.com

[2] J. Burns. Intent Fuzzer, https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx, 2009.

[3] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer. An Empirical Study of the Robustness of Inter-component Communication in Android. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2012.

[4] F. J. Hui Ye, Shaoyin Cheng, Lanbo Zhang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proc. of International Conference on Advances in Mobile Computing & Multimedia (MoMM)*, pages 68–74, Vienna, Austria, 2013. ACM.

[5] R. Sasnauskas and J. Regehr. Intent Fuzzer: Crafting Intents of Death. In *Proc. of the Joint International Workshop on Dynamic Analysis(WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5, San Jose, CA, 2014. ACM.

[6] M. P. Roee Hay, Omer Tripp. Dynamic Detection of Inter-application Communication Vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, New York, NY, USA, 2015. ACM.

[7] Wu Tianjun and Yang Yuexiang, Crafting Intents to Detect ICC Vulnerabilities of Android Apps, *the 12th International Conference on Computational Intelligence and Security*, pages 557–560, Wuxi, China, December 16-19, 2016.

[8] D. S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[9] Hwacha, a Flexible Intent Fuzzer with an Automatic Tally of Failures for Android, http://swlab.jnu.ac.kr/paper/hwacha.html

[10] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang, Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services, *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, pages 361–370, Los Angeles, CA, USA, December 7-11, 2015.

[11] Huan Feng and Kang G. Shin, BinderCracker: Assessing the Robustness of Android System Services, CoRR(Computing Research Repository), abs/1604.06964, 2016.

[12] Shauvik Roy Choudhary, Alessandra Gorla , and Alessandro Orso, Automated Test Input Generation for Android: Are We There Yet?, *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, Washington, DC, USA, November 9-13, 2015.

[13] Aimin Zhang, Yi He, and Yong Jiang, CrashFuzzer: Detecting Input Processing Related Crash Bugs in Android Applications, *the 35th IEEE International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, Las Vegas NV, USA, December 9-11, 2016.

[14] Hossain Shahriar, Sarah North, and Edward Mawangi, Testing of Memory Leak in Android Applications, *the 15th International Symposium on High-Assurance Systems Engineering (HASE'14)*, pages 176–183, Miami, Florida, USA, January 9-11, 2014.

[15] D. Amaltano, A. Fasolino, and P. Tramontana, A GUI Crawling-based Technique for Android Mobile Application Testing, *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 252–261, Washington, DC, USA, March 21-25, 2011.

[16] D. Amaltano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, Using GUI Ripping for Automated Testing of Android Applications, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, pages 258–261, Essen, Germany, September 3-7, 2012.

[17] Nariman Mirzaei, Automated Input Generation Techniques for Testing Android Applications, Ph.D. Thesis, Dept. of Computer Science, George Mason University, Summer Semester 2016.

[18] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk, Automatically Discovering, Reporting and Reproducing Android Application Crashes, *IEEE International Conference on Software Testing, Verification and Validation(ICST'16)*, pages 33-44, April 10-15, 2016.

[19] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-application Communication in Android. *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011.

[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities, *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, pages 229–240, Raleigh, North Carolina, USA, October 16-18, 2012.

[21] Andrea Avancini and Mariano Ceccato, Security Testing of the Communication among Android Applications, *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, pages 57–63, San Francisco, CA, USA, May 18, 2013.

[22] Vaibhav Rastogi, Yan Chen, and William Enck, AppsPlayground: Automatic Security Analysis of Smartphone Applications, *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, pages 209–220, San Antonio, Texas, USA, February 18-20, 2013.

[23] H. D. Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou. IntentFuzzer: Detecting Capability Leaks of Android Applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, pages 531–536, New York, NY, USA, 2014. ACM.

[24] Sébastien Salva and Stassia R. Zafimiharisoa, APSET, an Android aPplication SEcurity Testing Tool for Detecting Intent-based Vulnerabilities, *Interniation Journal of Software Tools Technology Transfer*, Vol.17, No.2, pp.201–221, Springer-Verlag, Berlin, Heidelberg, April, 2015.

[25] Davide Ghio, Mariano Ceccato, and Andrea Avancini, Identifying Android Inter App Communication Vulnerabilities Using Static and Dynamic Analysis, *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 255–266, Austin, Texas, USA, May 14-22, 2016.

[26] 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, *Proceedings of the 35th ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 259–269, New York, NY, USA, June, 2014.

[27] Dart, https://github.com/f2prateek/dart.

[28] Jackson, https://github.com/FasterXML/jackson.

[29] Myungpil Ko, A Design and Implementation of Intent Specification Language for Robust Android Apps, MS. Thesis, Yonsei University, Wonju, Korea, August 2015 (Written in Korean).

[30] https://github.com/secure-software-engineering/DroidBench