A Practical Intent Fuzzing Tool for Robustness of Inter-Component Communication in Android Apps

Kwanghoon Choi[1], Myungpil Ko[2] and Byeong-Mo Chang[3]
[1] Dept. of Electronics and Computer Engineering, Chonnam National University
Gwangju, 61186 – Republic of Korea
[e-mail: kwanghoon.choi@jnu.ac.kr]
[2] Computer&Telecommunication Engineering Division, Yonsei University
Wonju, Gangwon-do, 26493 – Republic of Korea
[e-mail: myungpil.ko@yonsei.ac.kr]
[3] Dept. of Computer Science, Sookmyung Women's University
Seoul, 04310 – Republic of Korea
[e-mail: chang@sookmyung.ac.kr]
*Corresponding author: Byeong-Mo Chang

Received August 1, 2017; revised November 26, 2017; accepted May 3, 2018; published September 30, 2018

Abstract

This research aims at a new practical Intent fuzzing tool for detecting Intent vulnerabilities of Android apps causing the robustness problem. We proposed two new ideas. First, we designed an Intent specification language to describe the structure of Intent, which makes our Intent fuzz testing tool flexible. Second, we proposed an automatic tally method classifying unique failures. With the two ideas, we implemented an Intent fuzz testing tool called *Hwacha*, and evaluated it with 50 commercial Android apps. Our tool offers an arbitrary combination of automatic and manual Intent generators with executors such as ADB and JUnit due to the use of the Intent specification language. 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 what is unknown so far. We also measured execution time for Intent fuzz testing, which has been rarely reported before. Our tool is practical because the whole procedure of fuzz testing is fully automatic and the tool is applicable to the large number of Android apps with no human intervention.

Keywords: Android, Intent, Vulnerability, Robustness, Fuzz

This research 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 Young, and Jisun Choi for their contributions on our Intent fuzzing tool.

http://doi.org/10.3837/tiis.2018.09.008 **ISSN : 1976-7277**

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 reached 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, though the component structure is good at modularity and reuse, which should 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 **Fig. 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: "*android.intent.action.INSERT*" (or "*INSERT*" in short) for creating a new memo and "*android.intent.action.EDIT*" (or "*EDIT*" in short) 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   = "No title";
       content = "Type your memo";
    }
    // Display a title and a content
  }
}
```

Fig. 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 invocations of *action.equals* will throw NullPointerException. Second, if any actions other than "*INSERT*" and "*EDIT*" are specified, displaying the title and content will cause to throw NullPointerException. This is because *title* and *content* become NULL on the other actions. Third, if any Intent with "*EDIT*" action misses one of values for two keys "*title*" and "*content*", the same problem will happen: the invocation of *getStringExtra* for the missing key/value will return NULL. Fourth, if any value for the two keys is set to be, say, an integer (a value of the incorrect type), the invocation of *getStringExtra* method for the incompatible key/value will return NULL, causing the same problem.

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 sticks to one's own Intent generation strategy. One cannot use the other Intent fuzzer's strategy without changing the implementation, limiting the capability to detect Intent vulnerability to one's own strategy. Null IntentFuzzer [2] set 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], some static and dynamic analyses had been employed to construct Intents more relevant to what Android apps deal with.

Second, the researches on Intent fuzz testing have rarely reported 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 would be more serious when millions of Android apps needed to be tested such as in Android Marketplace.

To address 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. According to the specifications, a generator will produce Intents, which an executor will take for testing.

For the second problem, we 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.

Based on the two new ideas, we have implemented a fully automatic Intent fuzzing tool, which we believe is the first practical one of the existing tools. We have evaluated 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. The tool and the experiment result are available in a companion web site [9]

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 fuzzing 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. Also, we have shown that the method can reduce much efforts on manually analyzing Android crash logs to identify different failures.
- The two new ideas mentioned above are not solely for a particular tool, but they are universally applicable to all Intent fuzzing 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 an Intent fuzzing tool, and have demonstrated the effectiveness of the two ideas by applying it to 50 commercial Android apps and finding more than 400 unique Intent vulnerabilities including one unknown so far.
- Our Intent fuzzing tool is practical because the whole procedure of fuzz testing is fully automatic and the tool is applicable to the large number of Android apps with no human intervention.

In Section 2, we discuss related work on Intent fuzz testing for Android apps. Section 3 presents what is Android Intent vulnerability. Section 4 introduces our proposal on Intent specification language. In Section 5, we describe the design and implementation of our fully automatic Intent fuzzing tool in detail. In Section 6, we show experiment results with commercial Android binary apps. In Section 7, we compare our tool with the existing ones. Section 8 concludes.

2. Related Work

Null IntentFuzzer [2] was the first Intent fuzzing tool to test Intent vulnerability for 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 Null IntentFuzzer 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, and random extra data. It was reported that JJB is a semi-manual approach. When a system alert is generated due to application crash, a user is involved in closing the alert dialog boxes. When an Activity is started as a new task, JJB cannot close it in an automatic manner since it is an Android app. In addition to these, JJB also 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 an Android app configuration file called *AndroidManifest.xml*, it built pieces of abnormal audio and video data for testing Activities. 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 and inter-procedural CFG analysis was employed to extract

the structure of Intents that each target component expects. The analysis traverses Dalvik bytecode instructions to collect calls to Intent's getter/setter methods and 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 [10] 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, target components were executed with these fuzzed Intents, and both code coverage and crashes due to exceptions were monitored. This research thus contrasts to the two previous researches [3][4] where static analysis is merely 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-based component communication, including Java crash, Fragment injection, UI spoofing, and Cross-application scripting (XAS). The purpose of using this tool was not only to detect Java crash as JJB, DroidFuzzer, and IntentFuzzer had aimed at, but it was also to discover other kinds of vulnerable Android apps. For example, it detected vulnerable Android apps 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. It was applied to Android apps from DroidBench (https://github.com/secure-software-engineering/DroidBench) and Google Play, and the number of vulnerabilities detected with this tool was compared with those by IntentFuzzer [5] and Null IntentFuzzer [2].

Besides Intent vulnerability associated with the robustness of Android apps, there have been researches on Intent vulnerability associated with security and privacy issues such as personal data loss, phishing, and other unexpected bad behavior. For example, Kun Yang et al's Intent fuzzer [11] showed a dynamic Intent fuzzing mechanism to uncover violations of permission model for Activity and Service.

3. 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 the Android app.

However, many misuses of Intent have been reported to cause Android apps crash, called *Intent vulnerability*, in [2][3][4][5][6][7]. 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.

Although the problem of Intent vulnerability is serious, there is no good mechanism yet to verify the well-formedness of Intents and to issue a warning for Intent vulnerability in Android apps. Programmers should find out the 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 **Fig. 1**, they 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. Another reason of difficulty comes from *implicit Intent* with no target component specified so that one can verify validity of implicit Intents only after a target component for the Intents is determined. As a result, Intent vulnerability in the code remains silent in compile-time, and then it arises in runtime, making it difficult to identify misuses of any Intents in Android apps early. This is a serious problem of the Android ICC (Inter-component communication) design.

We therefore approach this problem of Intent vulnerability by Intent fuzzing based testing, which dynamically executes Intents to see if Intent vulnerability is exposed.

4. The Intent Specification Language

Before we explain our Intent fuzz testing tool, this section introduces a specification language for describing the structure of Android Intent as shown in **Fig. 2**. Let us begin with an example of Intent specification for the structure of Intents that Note Activity in **Fig. 1** is supposed to receive, 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 }
```

An Intent specification is a sequence of fields surrounded by { and }, and it can be composed 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 for 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 form of ground Intent specification:

```
{ 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 . We design the Intent specification language by modeling the structure of Intent class in Android platform. An Intent specification includes a target component (*cmp*), an action (*act*), data (*dat*), a type (*typ*), and 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].

```
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.)
```

Fig. 2. The Intent Specification Language

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 vary 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 specify concrete values for the field as well.

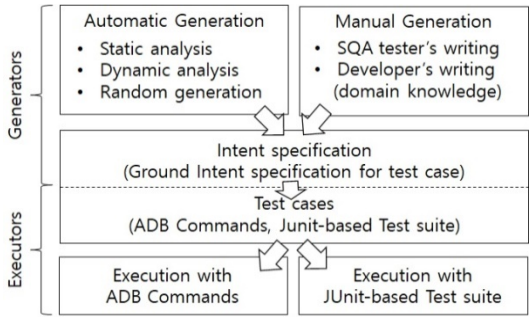

Fig. 3. Decoupling Generators from Executors by Intent Specification Language

The Intent specification language allows a modular structure of Intent fuzzing tool by decoupling generators of Intent test cases from the associated executors. It becomes very flexible to combine a generator with an executor in many ways, as depicted in **Fig. 3**. This structure enables a machine to generate automatically specifications for Intent-based 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 can make use of static analysis and dynamic analysis with random generation. Some static 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. For example, design documentations for Android projects can offer such domain knowledge.

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 fuzzing tool 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 **Fig. 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.

5. A Flexible Intent Fuzzing Tool with an Automatic Tally of Failures

Fig. 4 describes the structure of a fully automatic 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 three main steps between installation and uninstallation of an APK file are as follows. First, this tool automatically constructs Intent specifications from the component configuration of the input APK file. Second, it maps them onto ADB commands to execute on the installed Android app, and it receives the test execution logs via another Android SDK tool called *logcat* [1]. Third, it automatically filters the duplicates of the logs out to write a report in Microsoft Excel format for review. We will explain each step in the following sections.

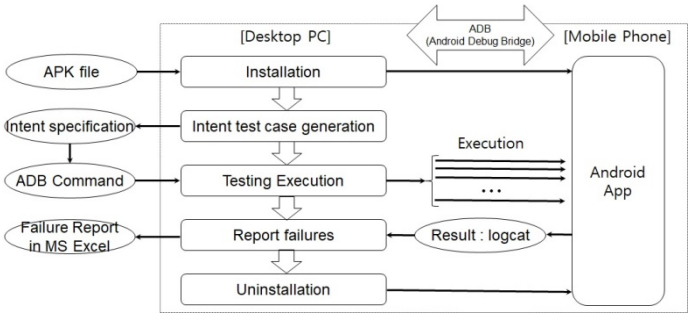

Fig. 4. The Architecture of Our Intent Fuzzer

5.1 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.

5.1.1 Automatic Construction of Intent Specifications

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 included. Each declaration of an Android component tells a Java class name of the component. The declaration also provides information on Intents to activate the component with. This information 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.

```
<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>
```

Fig. 5. An Example of AndroidManifest.xml

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

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, 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 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. For example, 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.

5.1.2 A Fuzzing Strategy for Generating Ground Intent Specification

Once an Intent specification is constructed for each Android component, what our tool does next is to generate ground Intent specifications that represent executable Intent test cases such as ADB commands or JUnit-based test suite. In the following, we will discuss a fuzzing strategy for refining given Intent specifications in order to generate ground ones, which can be immediately mapped into executable Intent test cases. Thus the Intent specification language connects Intent test case generators to the executors of Intent test cases smoothly with a fuzzing strategy.

We design a fuzzing strategy by observing the characteristics of how an Android component handles incoming Intents. According to our experience, most of Android components start with a series of conditional statements branching according to the action name of an incoming Intent, as shown in **Fig. 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. Also, an extra Intent data field (*dat*) is 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. But the action name is not necessarily the same as 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 and new extra data are introduced with type (*typ*).

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 it has 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.

5.2 Executing Intent Test Cases via ADB Commands

One form of Intent executors in our tool is the ADB interface with ADB commands into which we transform ground Intent specifications. Three ADB commands are transformed from one compatible and two incompatible ground Intent specifications explained in the previous section, 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.

Once a set of ADB commands is 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 in the logs some textual patterns to be explained below. It writes all Android logs and the analysis result in a result file.

The textual patterns that our tool uses to judge a failure from Android logs are “ActivityManager: Force finishing activity”, “AndroidRuntime: Shutting down VM”, and “ActivityManager: Process pid **** has died” for Activity, Service, and Broadcast Receiver component types, respectiveldy.

Our tool is designed to write all Android logs and the analysis result in a Microsoft Excel file as shown in **Fig. 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.

E	com.chbreeze.jikbang4a	AndroidRuntime	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.chbreeze.jikbang4a	AndroidRuntime	com.chbreeze.jikbang4a.support.WebViewSupport$WebViewClient.handleZigbangU
E	com.chbreeze.jikbang4a	AndroidRuntime	at com.chbreeze.jikbang4a.UriActivity.onCreate(UriActivity.java:17)
E	com.chbreeze.jikbang4a	AndroidRuntime	at android.app.Activity.performCreate(Activity.java:6020)
E	com.chbreeze.jikbang4a	AndroidRuntime	at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1105)
E	com.chbreeze.jikbang4a	AndroidRuntime	at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2284)
E	com.chbreeze.jikbang4a	AndroidRuntime	... 10 more
W		ActivityManager	Force finishing activity com.chbreeze.jikbang4a/.UriActivity

Fig. 6. An Example of Failure Logs in Microsoft Excel

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 continue to get any Android logs further. This will prevent us from running our tool over a large number of Android apps 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 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. This auxiliary mechanism allows us to have finished our experiment with 50 real-world Android apps in a batch run of our tool successfully without stopping in the middle.

5.3 Automatically Reporting Failures Without Duplication

A naive way of reporting failures 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 is only about 23%. Since the fuzz testing is very likely to produce many same failures, it will demand much efforts in classifying different failures. Hence, it is required to classify failures into sets of the similar failures automatically for reduction of such efforts in the post-classification.

Our tool employs a procedure of grouping 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, of which each index set tells occurrences of a unique 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]. 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 LCS is not necessarily contiguous in the two sequences.

In the LCS algorithm, each failure log such as one shown in **Fig. 6** is converted to a character sequence obtained by concatenating rows in the log before the algorithm is applied.

Our algorithm initially lets each failure log have an index set to itself. For each pair 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 of 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. 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.

Due to the nature of fuzz testing, the existing Intent fuzzing tools must produce many duplicate failures as well. The proposed classification algorithm could be also applied together with the tools to reduce efforts on classifying duplicate failures manually.

6. Evaluation

We present an experimental result on applying our Intent fuzzing tool to 50 popular Android apps downloaded from Google Play where no source code is provided. The Android apps are Facebook, Instagram, Kakao Talk, Naver, CGV, and so on. The detailed information about Android app names, versions and DEX binary code sizes are available in [9].

The source code of our tool, all Android APK files used in the experiment, and the result data are also available in our companion web site [9].

6.1 Failure Counts Due to Intent Vulnerability

First, our experiment strongly supports that the problem of Intent vulnerability on Android apps is serious: the two third of Android apps on the experiment experienced abnormal crashes due to Intent vulnerability. Our Intent fuzzing tool discovered total 409 different failures due to Intent vulnerability over 50 Android apps as shown in **Fig. 7**. On average, it is eight failures per Android app.

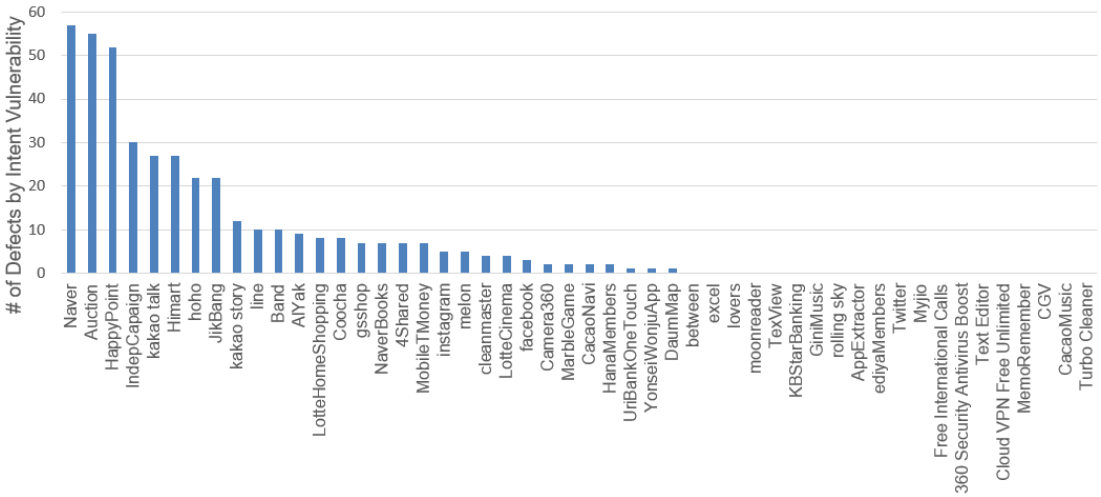

Fig. 7. The Number of Discovered Failures Due to Intent Vulnerability

Second, the ratio of failure counts over binary code sizes is found to be a simple and effective criterion to classify robust and weak Android apps in terms of Intent vulnerability. **Fig. 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.

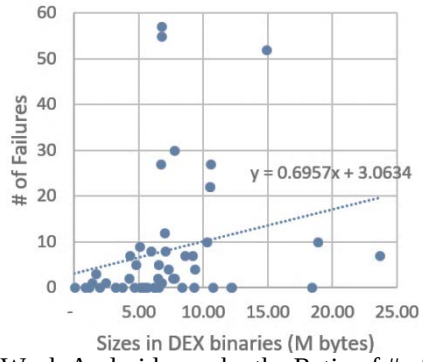

Fig. 8. Identifying Robust/Weak Android apps by the Ratio of # of Failures and Binary Sizes

The seven Android apps above the dotted linear line in **Fig. 8** are weak ones based on the ratio of failure counts over Android binary code sizes. Actually, they are the top seven Android apps in terms of the number of discovered failures as shown in **Fig. 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 very robust ones: 4shared (7 failures/23.67MB), Band (10 failures/18.87MB), and Twitter (no failures/18.41MB).

6.2 Automatic Classification of Failures in Android Logs

Our tool is very effective in classifying duplicate failures from Android logs, judging from the ratio of the number of failure groups to that of all failures as shown by **Fig. 9**. It found 79.2% of all failures in a single Android app reappear on average, and so it 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.

Fig. 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 (7th entry of the graph in **Fig. 9**) to HanaMembers, our tool successfully removed 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 failure with 1 group), DaumMap (1 failure with 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.

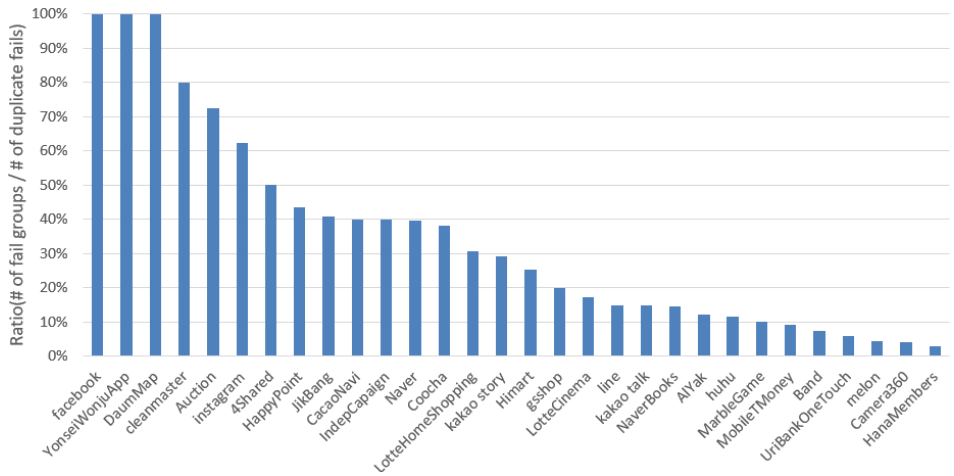

Fig. 9. Ratio of the number of failure groups to that of all failures

Only the fifth Android app, Auction, with 76 failures identified into twelve groups worth a further analysis. The grouping result is G_0={0,1,2,3,4,5,6,8,11,12,13,18,20}, 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 consist 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. In the case of the Android logs #23 and #24 of Auction, we confirm that they are different from each other by inspection. However, it could be more likely to miss detecting duplicates as Android logs are large: the same lines in the duplicates may appear in different order. The failure classification method could be improved in this respect by more discerning method such as machine learning algorithms.

6.3 Execution Time for Intent Fuzz Testing

We evaluated our tool in terms of the execution time for Intent fuzz testing, which have been rarely discussed by the existing researches. 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.

Fig. 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.

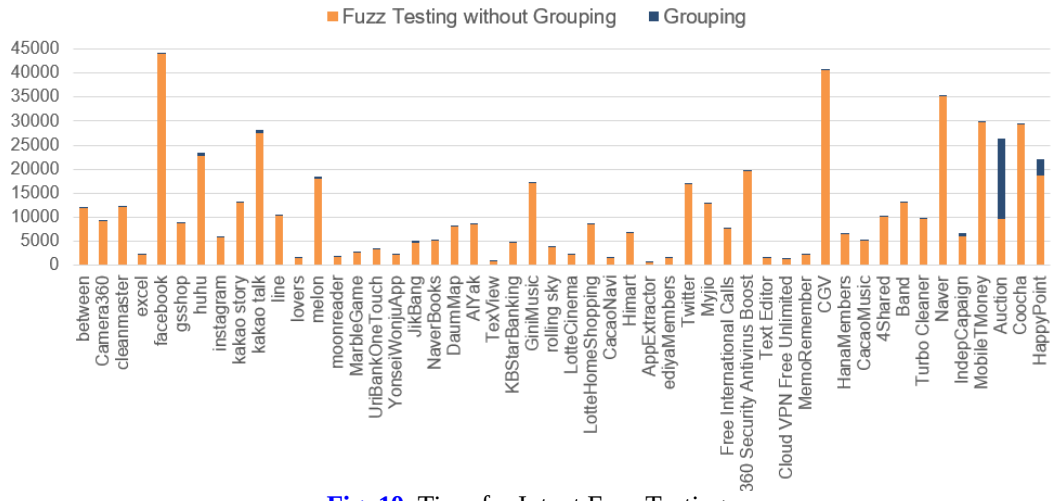

Fig. 10. Time for Intent Fuzz Testing

The time for Intent fuzz testing on a single Android app is found to be proportional to the number of Intent test cases. **Table 1** tells us that for the fifty Android apps, our tool automatically constructed 3,408 Intent specifications (68.2 per an Android app on average) by the method explained previously. The full detail of the table is available in [9]. The Intent specifications expanded to 102,240 Intent test cases (ground Intent specifications) by generating 30 Intent test cases per each Intent specification. We tested 2,045 Intent test cases over a single Android app on average, and each Intent test case required about 5 seconds due to the reason explained in Section 5.2. 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.

Table 1. An Experiment Result for Intent Vulnerability

	# of Intent Specs	# of Intents	# of Failures	# of Groups (# of Unique Failures)
Total	3,408	102,240	1,797	409

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 **Fig. 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) did. The reason is that the length of the failure logs was relatively long and the number of the failure log groups was 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. Such measurement of execution time for testing Intent vulnerability 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 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.

In summary, this testing time analysis suggests that some consideration should be taken for efficiency of Intent fuzz testing.

6.4 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 tool discovered an interesting failure that the other researches had not reported before.

Table 2 shows a summary of statistics on kinds of exceptions due to Intent vulnerability discovered by our Intent fuzz testing tool on the fifty Android app.

Table 2. A Statistics on Exceptions Causing Intent Vulnerability

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

The most frequent root cause of Intent vulnerability was 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, some 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, revealed a serious mismatch between the package name of a class and its declaration on AndroidManifest.xml. One of the associated failure logs was as follow:

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

where the mentioned class BarcodeRecognitionActivity was found in differently named package com.nhn.android.search.ui.recognition. But the wrong package name was declared in the AndroidManifest.xml of the Android app, and so it was legal to attempt to invoke BarcodeRecognitionActivity with this wrong package name. This was 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, were caused by some extra field set with values of types different from integer. To prevent this exception, validity of values from Intents should be verified 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 was 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 was found to be thrown. The associated method was accessible because it was declared as public. However, according to the documentation in Android source code by Google, it is intended that the method is internally used. Without further information on the relevant Android apps, it was 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.`
  - `at com.google.inject.internal.util.$Finalizer.startFinalizer(Finalizer.java:77)`

For the second entry in **Table 2**, the relevant failure logs showed few clue for us to decipher the causes of Intent vulnerability partly because there was no source available for Android binary apps for the experiment.

7. Discussion

7.1 Comparison with Existing Intent Fuzzing Tools for Detecting Crashes

In **Table 3**, we compare our Intent fuzzing tool, named *Hwacha* [9], with the existing tools for detecting crashes due to Intent vulnerability [2][3][4][5][6][7]. 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.

Table 3. Comparison with the Existing Intent Fuzzing Tools

	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 [10] 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. To point out this reason, we have two marks ‡ in the static and dynamic analysis columns, and therefore Hwacha has Y in the Generic Provision column of the table. 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 have 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 automatically, for example, as is explained in Section 5.2. 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 getting stuck, 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, there is no previous work on attempting to removing duplicate failures in Intent fuzz testing. Due to the nature of fuzz testing, one can try many different Intent test cases resulting in 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 for removing duplicate failures is 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.

Hwacha has a capability to generate JUnit test code based on Android testing framework as well, which is useful for programmers with source code. When an Intent fuzzing 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. Android Studio can run the generated JUnit test code. 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 nor dynamically in run-time by Android platform.

The second author of this paper in his master's thesis [12] designed a runtime assertion library using Intent specification to describe the structure of incoming well-formed Intents and to specify how to handle malformed Intents.

```
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 Fig. 1
    }
    ...
  }
```

This example of the runtime assertion library starts from the example of Note Activity in **Fig. 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 runtime verification of Intents.

Maji et al [3] suggested 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. 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 using interface definition languages (IDL).

Dart & Henson (https://github.com/f2prateek/dart) is a library for Android which uses annotation processing to generate code that does direct field reading and assignment of extras

of Intents. Jackson (https://github.com/FasterXML/jackson) is a suite of data-processing tools for Java to serialize structured data such as XML documents and to do the reverse, checking validity. It has been developed for XML representation but could be applied to Intents.

8. Conclusion

We have developed a fully automatic Intent fuzzing 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 caused by Intent vulnerability, including what has never been reported before. 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.

Of the existing Intent fuzz tools, ours is the first practical tool. Because the whole procedure of fuzz testing is fully automatic, the tool will be suitable for Android app developers who are not much familiar with Android ICC design. Because the tool itself is stable enough for mass application with no human intervention, it will be useful for SQA team to test Android apps repeatedly, and for marketplace managers to examine a large number of Android apps.

References

[1] Google, “Android Developers,” 2012. [Online]. Available: Article (CrossRef Link).
[2] J. Burns, “Intent Fuzzer,” *2009*. [Online]. Available: Article (CrossRef Link).
[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 *Proc. of Proceedings of the International Conference on Dependable Systems and Networks*, pp. 1-12, June 25-28, 2012. Article (CrossRef Link)
[4] H. Ye, S. Cheng, L. Zhang, and F. Jiang, “DroidFuzzer : Fuzzing the Android Apps with,” in *Proc. of Int. Conf. Adv. Mob. Comput. Multimed.*, pp. 2–4, December 2-4, 2013. Article (CrossRef Link)
[5] R. Sasnauskas and J. Regehr, “Intent fuzzer: crafting intents of death,” in *Proc. of the 2014 Joint Int'l Workshop on Dynamic Analysis and Software and System Performance Testing, Debugging, and Analytics - WODA+PERTEA 2014*, pp. 1–5, July 22, 2014. Article (CrossRef Link)
[6] R. Hay, O. Tripp, and M. Pistoia, “Dynamic detection of inter-application communication vulnerabilities in Android,” in *Proc. of Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pp. 118–128, July 13-17, 2015. Article (CrossRef Link)
[7] W. Tianjun and Y. Yuexiang, “Crafting Intents to Detect ICC Vulnerabilities of Android Apps,” in *Proc. of Int'l Conference on Computational Intelligence and Security*, pp. 16–19, December 16-19, 2016. Article (CrossRef Link)
[8] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Commun. ACM*, vol. 18, no. 6, pp. 341–343, June, 1975. Article (CrossRef Link)
[9] K. Choi, “Hwacha, a Flexible Intent Fuzzer with an Automatic Tally of Failures for Android.” [Online]. Available: Article (CrossRef Link).
[10] S. Arzt *et al.*, “FlowDroid : Precise Context , Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps,” in *Proc. of 35th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, pp. 259–269, June 9-11, 2014. Article (CrossRef Link)
[11] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, “IntentFuzzer: detecting capability leaks of android applications,” in *Proc. of Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 531–536, June 4-6, 2014. Article (CrossRef Link)

[12] M. Ko, "A Design and Implementation of Intent Specification Language for Robust Android Apps," *Master Thesis*, Yonsei University, Wonju, Korea, August 2015.

Kwanghoon Choi received his Ph.D degree from KAIST in 2003. He is currently an associate professor at Chonnam National University. His research areas include programming languages, type systems, compilers, software engineering, and mobile computing.

Myungpil Ko received his MS degree from Yonsei University, Wonju in 2015. His research areas include programming languages, testing, software engineering, and mobile computing.

Byeong-Mo Chang received his Ph.D degree from KAIST in 1988. He is currently a professor at Sookmyung Women's University. His research areas are programming languages, program analysis and verification, and software security.