Demonstration Paper

Intent Specification Language and Its Tools for Testing the Vulnerability of Android Components

Myung-Pil Ko
Yonsei University, Wonju, Korea
myungpil.ko@yonsei.ac.kr

Kwanghoon Choi
Yonsei University, Wonju, Korea
kwanghoon.choi@yonsei.ac.kr

Byeong-Mo Chang
Sookmyung Women's University, Korea
chang@sookmyung.ac.kr

Abstract

We design an Intent specification language and develop its tools for testing Android components to help to develop robust Android programs. Every Android program has the structure of components communicating among themselves with messages called *Intents*. Recent researches have pointed out that many Android programs pose vulnerability due to the absence of any procedure of verifying if intents miss any fields or if they carry ill-formed values. The proposed specification language enables programmers or even software to write the shape of intents used in Android component communication. We call such a description an *intent specification*. The companion tools generate testing artifacts from the intent specification automatically, and also help to run the generated testing artifacts to evaluate the vulnerability of each Android component. Currently, our tools provide the automatic generation of two types of testing artifacts: ADB (Android Debug Bridge) commands and Android JUnit test cases. This framework on the Intent specification language will greatly ease the burden of testing the vulnerability of Android components. The proposed idea of the Intent specification language is new and it is very flexible in integrating with any new kind of testing artifacts. This contrasts with the related research work. We show a preliminary testing result in an application of our tools to real-world Android programs in our experiment.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords Android, Vulnerability, Specification, Testing, Tool, Component

1. Introduction

Every Android program is a Java program with Android APIs. Android is Google's open-source platform for mobile devices, and it provides the APIs (Application Programming Interfaces) necessary to develop applications for the platform in Java [1]. An Android program has the structure of components such as activities, services, and broadcast receivers, where the components communicate among themselves by sending messages called *Intents*.


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

Figure 1. Android Program Example

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 above, the *Note* activity accepts two actions: *"INSERT"* for creating a new memo and *"EDIT"* for editing an existing one.

- When 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`.
- When 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 $intent.getStringExtra$ returns `NULL` due to the missing value.
- When any value of the two keys is set to be, say, an integer (not a string), an invocation of $intent.getStringExtra$ method with the corresponding key will return `NULL` because of the incorrect type of the value.

Recent researches [2–5] have discovered how vulnerable Android components are due to Intents. In experiments [2], they generated random and semi-valid intents and tested how components handle these exceptional conditions by extending the basic

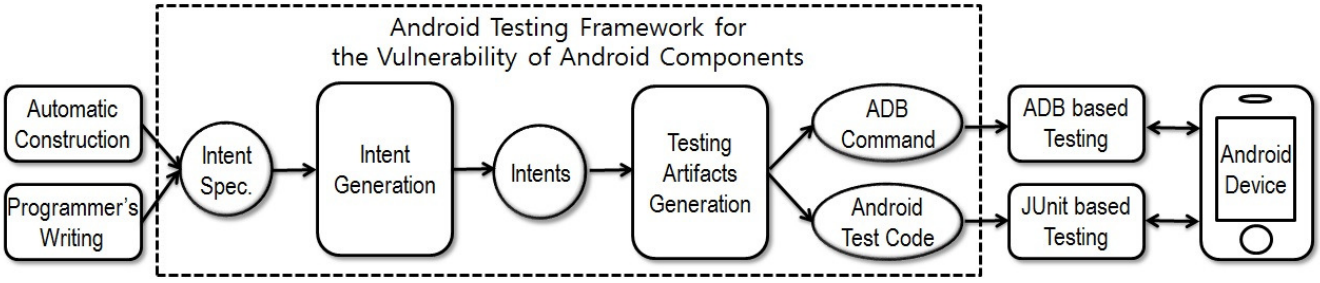

Figure 2. Architecture

Intent fuzzer [3]. In particular, they focused on uncaught exceptions, because they result in the crashes. In the experiment, 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 are 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.

A combination of static analysis and random fuzzing was also proposed to dynamically test Android applications in [4]. A path-insensitive, inter-procedural CFG analysis is employed to automatically extract the expected intent structure that a component is expecting to receive. A set of intents are generated with the static 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 are monitored.

When input from Intents are not validated sufficiently, a malformed Intent delivered to a receiver may expose attack surfaces as pointed out by [5]. For example, unauthorized receipt of an implicit Intent can be made by malicious component, and Intent spoofing can be made, by which a malicious application sends an Intent to an exported component that is not expecting Intents from that application. A static analysis tool like ComDroid detects statically these potential vulnerabilities in Android applications [5].

Although the existing research tools have addressed the vulnerability of Android components by static and dynamic analyses, they lack a common framework to express in a flexible way the shape information of intents, what we call *intent specification*, and to generate test artifacts in various testing contexts. This motivates us to design an *Intent specification language* and to develop its tools for testing the vulnerability of Android components. The proposed language allows programmers to express an important property of Android programs on intents. As an intermediary, it also facilitates multiple combinations of the ways of writing intent specification and those of testing artifacts generation.

In Section 2, we propose an Android testing framework based on an intent specification language and its tools. A typical scenario using the tools are described. We briefly discuss the technology used for implementation. Section 3 shows preliminary experiment results of a specification-based testing and an automatic testing using the tools with real Android programs. Section 4 concludes with an outline of missing features as a future work.

2. A Testing Framework for the Intent Vulnerability of Android Components

A graphical overview of a testing framework and its tools to demonstrate is shown in Figure 2. Under the framework, a typical scenario for testing the vulnerability of Android components is to write an intent specification for each component, to generate intents from it, to generate test artifacts from the intents, and to run the artifacts in each testing environment, as will be explained in the following.

An intent specification describes the shape or information of intents. For example, for intents that the example activity component *Note* accepts, one may write as follows:

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

The described intents have either *EDIT* action or *INSERT* action. The former action accompanies two extra strings identified by keys *title* and *content* with type name *String*, as shown by the example of the intent specification. Optionally, a specific instance of strings can follow the type name. A full detail on a grammar for the Intent specification language is available in [6].

The presence of a well-defined Intent specification language enables programmers or even software to write the shape of intents used in Android component communication.

- When developing an Android project, every programmer clearly knows what components accepts which intents. The programmer may write intent specifications based on the knowledge.
- When only an Android binary program is available, a software tool may construct potential intent specifications based on some meta information file (e.g., AndroidManifest.xml) in the binary.

Although our tools support only the above two ways of writing specifications now, any static analysis tools such as ComDroid [5] can be employed to help to write intent specifications efficiently.

Once an intent specification for an Android component is available, a tool in the framework automatically generates intents that can be used to activate the component for testing. The intent generation tool supports three levels of qualification on how much similar generated intents are to the intent specification: *compatible*, *shape-compatible*, and *random* intents.

Every compatible intent has the same fields and values as they are guided by the specification. Sometimes, it can include more fields than the specification. For example,

```
{ cmp = com.example.android/.Note
  act = android.intent.action.EDIT
  [ title = String "my title",
```

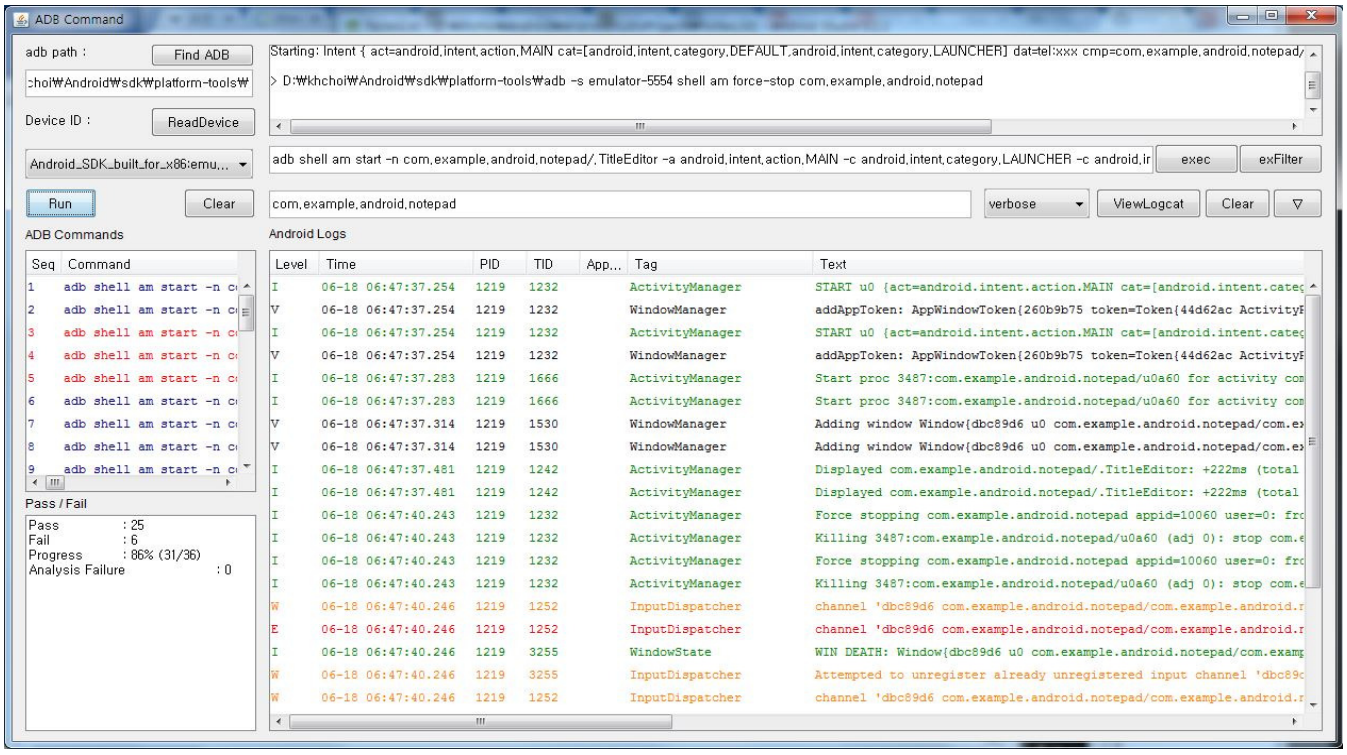

Figure 3. Screenshot: Running ADB Commands Generated from an Intent Specification

```
    content = String "my_content"]
  dat = qoFXwARtpfV–LNN }
```

where the field of key *dat* is supposed to have a URI as the name of data. The generated intent, however, has a malformed URI.

Every shape-compatible intent includes all the fields in the specification, but the field values may be generated randomly. It can also have more fields than the specification.

Random Intents are ones as the name stands for. They might not always follow the specification, as the intent fuzzing approach [2, 3] did.

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

where the categories field (*cat*) may be used for deciding a target component.

Now we are ready to generate testing artifacts. Our framework can generates ADB (Android Debug Bridge [1]) commands and Android JUnit test codes, automatically. Both of them are executable in their own testing environment.

For example, the compatible intent in the above example will give rise to an ADB command, as follows:

```
adb shell am start
  -n com.example.android/.Note
  -a android.intent.action.EDIT
  -d qoFXwARtpfV–LNN
  --es title "my_title"
  --es content "my_content"
```

The full exposition of ADB commands are available in [1].

One can issue ADB commands on a shell terminal to instruct an Android device to activate the specified component with the intent. Figure 3 shows our tool running multiple ADB commands in sequence for testing specified Android components, showing the results, and offering logs both by UI in real-time and by Microsoft Excel format for off-line reviews.

Intent specifications are also used for automatically generating Android JUnit test codes, as shown in Figure 4. For example, from the intent we discussed previously, the following code is generated.

```
package com.example.android;
...
public class NoteTest_0001
        extends ActivityUnitTestCase<Note>
{
 public NoteTest_0001()
   { super(Note.class); }
 public void testcase1() {
  Intent intent = new Intent();
  intent.setClassName("com.example.android",
                      "com.example.android.Note");
  intent.setAction("android.intent.action.EDIT");
  intent.putExtra("title", "my_title");
  intent.putExtra("content", "my_content");
  try {
    intent.setData(Uri.parse("qoFXwARtpfV–LNN"));
  }
  catch (Throwable t) { }
  startActivity(intent, null, null);
 }
}
```

where *ActivityUnitTestCase* is a class offered by an Android framework for writing Android test cases and suites [1].

This generated code is executable in the standard Android development environment (Android Studio [1]) where programmers can run the generated test codes and can review the testing results.

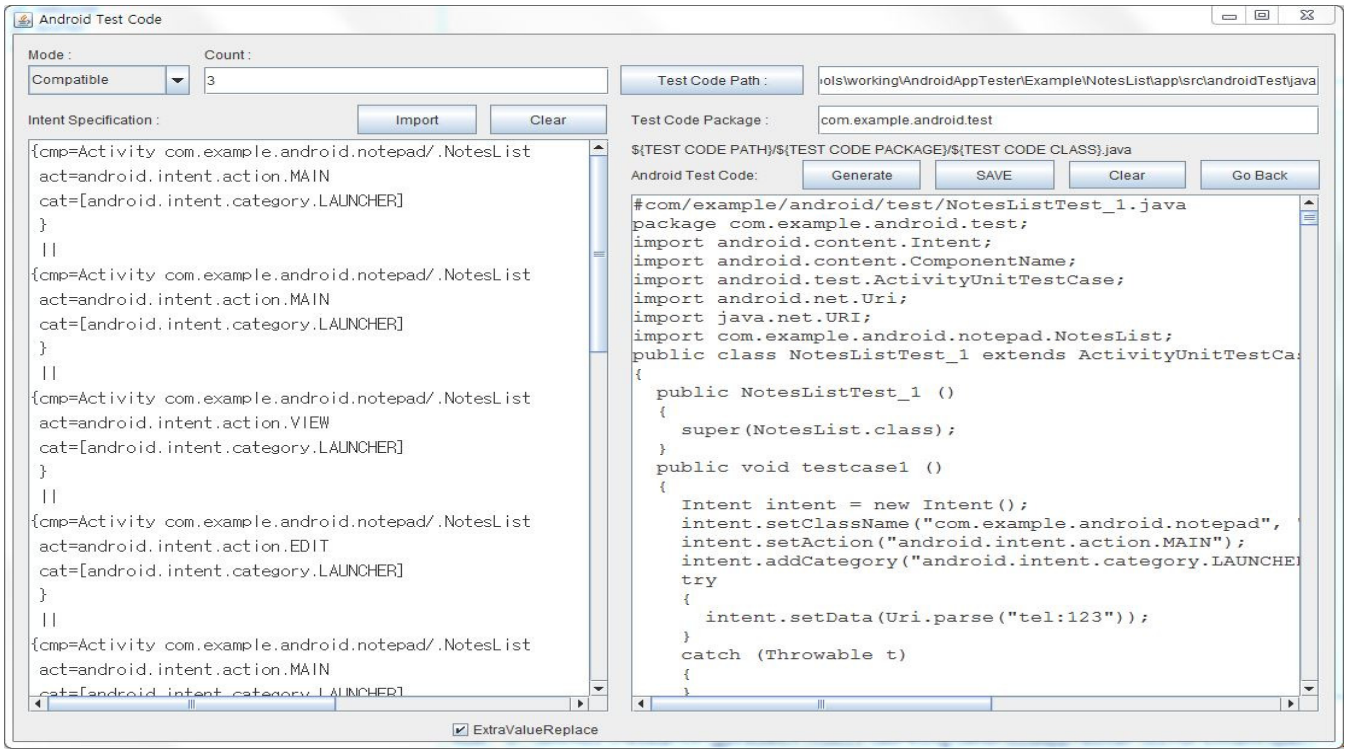

Figure 4. Screenshot: Generating Android JUnit Test Code from an Intent Specification

Further information on our tools is available in [6].

3. An Application: Detecting Vulnerable Apps in Android Market

This section reports a preliminary result on an application of our tools, assuming a scenario in an Android market that a manager may make use of our tools to identify quickly the vulnerability of Android apps to report to their developers. For each Android binary program, our tools construct an intent specification for each component of the program automatically. From the generated specifications, the tools generate as many intents and the corresponding ADB commands as the manager wants to test the components with.

We applied our tools to 10 real-world Android programs in the following table:

Android Apps	Intent Specs	Intents	Errors
B2N	38	1140	2
CGV Movie	121	3630	11
Facebook	153	4560	0
Naver	108	3240	0
Marble Game	8	240	0
Moon+Reader	6	180	0
AfricaTV	17	510	2
Yonsei Univ.	7	210	0
Woori Bank	12	360	0
Kakao Talk	91	2730	12

In summary, the tools generated 561 intent specifications and 16830 intents for testing [6]. We found that 4 Android programs terminate abnormally with exceptions due to the intent vulnerability. The exceptions are `NullPointerException`, `RuntimeException`, and `IllegalArgumentException`, which are all unchecked ones in Java. Although these results are preliminary, we believe that they show the significance of our framework.

4. Conclusion

In this paper, we propose an Intent specification language and develop its tools for testing the Intent vulnerability of Android components. This is a common framework to express in a flexible way the shape or information of intents used to activate Android components, and can support various combinations with any testing artifact generators including the two we developed.

We continue to enhance these tools, for example, by a facility to group the similar faults for better analysis reports and by more convenient ways to write intent specifications.

References

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

[2] 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 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp.1-12, IEEE Computer Society, Washington, DC, USA, 2012.

[3] Intent Fuzzer, https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx, iSEC partners, 2009.

[4] R. Sasnauskas and J. Regehr, "Intent Fuzzer; Crafting Intents of Death," In Proceedings of the Join International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), pp.1-5, ACM, San Jose, CA, USA,, 2014.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobySys), pp.239-252, ACM, New York, NY, USA, 2011.

[6] http://mobilesw.yonsei.ac.kr/paper/intent_spec.html