WHERE DOES YOUR INTENT GO AND HOW IT BEHAVES? A ROBUSTNESS STUDY OF INTENT REACHABILITY AND HANDLING IN ANDROID SYSTEMS

A Thesis

by

XU YAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Guofei Gu
Committee Members,	Yoonsuck Choe
	Jim Xiuquan Ji
Head of Department,	Dilma Da Silva

May 2015

Major Subject: Computer Science

ABSTRACT

In Android systems, inter-process communication relies heavily on intent, which can be understood as a message between apps. However, (i)intents can be hijacked when they are transmitted from senders to receivers. Even if received securely, due to developer's underestimation of intent data complexity, (ii)intents can cause exceptions in their receivers. An app is at the risk of losing response, and even crashing if it fails to handle the exceptions properly.

To deal with the two potential problems above, we added an Android framework-layer module to reject the installation of suspicious apps that may hijack intents during transmission. In addition, we proposed and implemented FuzzingDroid, a utility tool that generates various relevant intents to fuzz test publicly-accessible intent receivers in apps. The tool is important because it helps developers detect the weakness of their incoming intent handling code before they release their apps. At its core, FuzzingDroid relies on our instrumented Android framework-layer module to generate the variant parts in fuzzing intents. The outcome of using FuzzingDroid is pretty good: after analyzing 47 highly-downloaded apps from Google Play Store, 46 highly-downloaded apps from other popular online app markets, 45 core system apps from LG Nexus 5 and 32 core system apps from XiaoMi phone respectively, we found 49 of the total 170 apps were crashed due to various intent handling deficiencies. FuzzingDroid is also a very efficient tool. It takes about 1 minute to fuzz an app completely with only 5% increase in CPU utilization and 24MB increase in memory utilization.

DEDICATION

To my parents

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professor Guofei Gu for giving me the opportunity to work with him, who is the source of inspiration of this work. He teaches the most impressive courses in our department, and always has insightful thoughts about research. The experience of working with him will guide me in my future career. I would also like to thank Professor Yoonsuck Choe and Professor Xiuquan Ji for serving on my committee. I would like to extend my thanks to my colleagues at SUCCESS Lab, to Srinath Nadimpalli and Guangliang Yang for giving me valuable suggestions on my work, to Jialong Zhang, Visvanathan Thothathri for their great help on my thesis defense preparation. It is my fortune to work with these wonderful guys. Finally, I gratefully acknowledge my parents for their support and encouragement throughout my whole life.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

IPC is short for inter-process communication. In Android systems, an application runs in a dalvik virtual machine instance. Each dalvik virtual machine instance runs in a separate process. Therefore, Android IPC mechanism is critical for inter-application communication. There are three types of IPC mechanisms in Android: (i) Binder, which allows one app to call a routine in another app by identifying the method to invoke and passing arguments between apps, (ii) ASHMEM, which declares a part of memory that can be shared by apps, and (iii) Intent, which is used as a message by an app component to start another app component either in the same process or different processes. Intent is the most heavily-used IPC mechanism in Android systems.

The two primary pieces of information in an intent are receiver and data. In explicit intent, receiver is specified with package name and class name. In implicit intent, receiver is specified with an action. Action can be understood as a requirement that receiver must satisfy in order to be eligible to receive the intent. Each receiver in an app can declare what kind of intent it can deal with in the app's configuration file AndroidManifest.xml. Android system is responsible for resolving and dispatching implicit intents to their receivers. Data is optional depending on if a sender has information to share with a receiver. Normally, data can be represented in a form of key-value pair(referred as Extras in Android) or unified resource identifier to MIME type data, such as images, videos, etc. An example of implicit intent is *Action:*ACTION_VIEW—*Data:*tel:2144996577, it will be received by phone dialer with 2144996577 filled in.

Intent IPC mechanism can become an attack surface if used inappropriately. Two

well-known intent-based attacks are (i) Intent Spoofing: If an intent receiver does not restrict who may send it intents, then an attacker may forge and inject malicious data into it, even complete privileged operations using the intent receiver as a proxy (commonly referred as Privilege Escalation Attack[1]). (ii) Intent Hijacking: If an intent sender does not specify the receiver explicitly, then an attacker may intercept the intent in the middle and make the expected intent receiver unreachable. Intent Spoofing attack can be defended by checking intent sender's permission at intent receiver side. However, defending Intent Hijacking is more complicated.

1.1 Intent reachability study

The difficulty to launch Intent hijacking attack depends on the type of target intent receiver. If the receiver is an activity, attackers have to fabricate their malicious apps to be as similar to their target apps as possible. Like phishing attack, an attacker tries to confuse users when they select the app to complete from candidate list. If the receiver is a service or an ordered broadcast receiver, when multiple intent receivers can deal with the same implicit intent, Android systems will automatically start the most appropriate one following the "First Loaded, First Served" rule.

Most existing state-of-the-art approaches detect intent hijacking with static analysis. ComDroid[2], proposed by Chin et al., statically analyzes dalvik executable files of an Android app. Comdroid defines intent declaration statements as sources and APIs that are used to send an intent to receiver as sinks. Comdroid will issue a warning when it finds a special source-sink connection, where the source declares an implicit intent but the sink does not specify any permission that protects the implicit intent. CHEX[3], proposed by Long Lu et al., detects possible hijacking-enabled flows by conducting reachability test on customized system dependence graph. A common feature of these tools is they need to be downloaded, configured and run against an

app to decide whether the app is suspicious at intent hijacking or not.

Complementary to previous work on intent hajiacking detection, we prevent the threat from system's perspective by adding a new Android framework-layer module in Android Open Source Project (AOSP hereafter). Our solution does not put any burden on end users. In addition, we approach the detection of implicit intent hijacking from a new perspective. Simply speaking, our module rejects the installation of an app if it intentionally overstates its capability to deal with implicit intent. We will talk about this in detail in section 4 of the thesis.

1.2 Intent handling study

Even if an intent is received by its expected receiver, due to the dynamic nature of an intent object, the receiver may underestimate the complexity of data associated with the incoming intent and fail to resolve the intent properly. The failure may lead to app crash and even phone restart. Fuzzing test is suitable to detect this kind of bugs in apps because (i) Intent can be sent by any app at any time, (ii) Android system provides comprehensive APIs to associate various types of data with an intent. Therefore, developers can easily generate random and semi-valid intents and send them to target apps to test those apps' robustness when handling incoming intents. Existing state-of-the-art approaches to fuzz data associated with an intent during IPC rely on static analysis tools to collect what kind of data is expected in an intent. However, the preprocessing stage with static analysis is time-consuming. Besides, it can generate false positives sometimes.

Complementary to those works, we propose FuzzingDroid, the fastest automated testing tool that can find out and test each type of publicly-accessible intent receivers with whatever data they expect. Simply speaking, FuzzingDroid will intercept data extraction call from an intent at runtime and redirect the call to our intrumented

Android framework-layer module. Our module will be responsible for returning various data of expected type back to the intent receiver. With our design, what type of data an intent should carry is not necessary to be known beforehand at all. This can save a big amount of time during fuzzing process. With our FuzzingDroid, we tested 47, 46 highly-downloaded apps from Google Play Store and other popular online app markets respectively. We also tested 45 and 32 core system apps from Nexus 5 and XiaoMi phone respectively. From these, we found 28.8% tested apps contain at least one intent handling deficiency that can crash the app.

1.3 Contributions

The major contributions of this thesis are summarized as follows,

(1) We present an in-depth study of implicit intent reachability in Android systems. We observe that implicit intent hijacking can be triggered deterministically when an implicit intent is used for starting service or sending ordered broadcast in Android systems. Android systems will follow a "First Load, First Served" rule when selecting implicit intent receivers that are equally eligible.

(2) We implement an Android framework-layer module to reject the installation of suspicious apps and modify the logic when Android systems resolve an implicit intent from systemś perspective to defend intent hijacking.

(3) We propose FuzzingDroid, a novel and fastest fuzzing test tool to automatically trigger potential bugs in an app's publicly-accessible intent receivers by sending intents with random and unexpected data.

(4) An in-depth evaluation of FuzzingDroid with respect to its runtime performance and efficacy in bug detection.

1.4 Thesis outline

The rest of this thesis is organized as follows: Section 2 introduces related works. Section 3 introduces background knowledge of Androidś intent receivers and app configuration file. Section 4 describes our study on ensuring intent reachability to target receiver. Section 5 presents the design and implementation of FuzzingDroid. Section 6 summarizes our conclusions.

2. RELATED WORK

Aravind et al. proposed a system called Dynodroid[4] for generating motion and data inputs to mobile apps. Their tool can fuzz smartly by selecting program paths that have not been explored before and can allow human to generate inputs when necessary. However, Dynodroid restricts apps from communicating with other apps and reverts to the app under test upon observing such communication. Jesse Burns proposed a unprivileged Null Intent Fuzzer[5] that injects valid intents with the blank data field to other apps publicly-accessible components. This is the first fuzzer on intent. However, the test coverage is quite restricted since an intent from this fuzzer never contains any data at all. An empirical study of the robustness on inter-component communication by Maji et al.[6] extended the Null Intent Fuzzer by creating a set of valid and semi-valid intents with object fields selectively left blank. Besides, they are the first to add extra data field in their fuzzing intents. The Intent Fuzzer by Raimondas et al.[7] improves the work further by crafting intent with expected structure to cover more program paths. Their work relies on FlowDroid to collect intent extras data field information instead of using purely random data during fuzzing process. Hui Ye et al. proposed DroidFuzzer[8] to fuzz UIs with URI data(e.g., "video/*", "image/*", etc.). DroidFuzzer modifies the inner structure of expected MIME type data and sends the abnormal data to target UI. Their tool successfully crashes several popular video and audio players. However, MIME data is only one type of data an intent can carry and UI is only one type of intent receivers. Our FuzzingDroid can also fuzz commonly-used URI-based MIME type data, such as video, audio, image and URL data. Kun Yang et al. proposed IntentFuzzer[9] and fuzzed publicly-accessible intent receivers with properly-constructed intents. Their

fuzzing process is feedback driven. Their tool relies on the debugger log to collect appropriate intent extras infomation on each round. However, our FuzzingDroid is much more efficient at intent extras information collection and fuzzing intents can be constructed properly with much shorter time.

3. BACKGROUND

In this section, we provide a brief overview of which components in an Android application can send and receive intent (We refer those components that can receive intents from other components as intent receivers hereafter). Then we present how to determine which intent receivers an Android app contains. Finally, we introduce how to launch intent hijacking from attacker's perspective and what can cause intent receiver to crash briefly.

3.1 Intent receivers

An Android app is composed by components. Of the four types of component in Android apps, Activity, Service and BroadcastReceiver can send and receive messages to/from other components either in the same or different processes. Activity is user interface in foreground. Service is background task that performs long-running operations, such as downloading service, audio playing service, etc. BroadcastReceiver responds to system-wide broadcast announcements. E.g. A BOOT_COMPLETE message will be broadcasted when an Android device is turned on and ready to be used. All broadcast receivers that register as recipients of the BOOT_COMPLETE will be notified synchronously. Android systems also support ordered broadcast. An ordered broadcast will be dispatched to its receivers asynchronously according to their priorities.

In Android systems, the message between different components is an intent object. An intent must indicate its receiver and could contain data shared between sender and receiver. We focus on implicit intent[10] in our study. Instead of specifying receiver with full-qualified class name, implicit intent indicates a general action to perform. Android system service (Package Manager Service) will match the intent

against candidates that can fulfill the action to decide which receiver should be the destination.

3.2 App configuration file

Each Android app contains a configuration file called AndroidManifest.xml. It specifies package name, requested permissions, the components an app is composed of, etc. A component is public if its declaration sets the EXPORTED flag or includes at least one $\langle intent - filter \rangle$[11]. Only public intent receiver can be invoked by components from different apps.

For any public intent receiver that can receive implicit intent, it must contain at least one $\langle intent - filter \rangle$, which restricts incoming intents by action, data, category, or any subset thereof. Action field specifies a general operation to be performed. E.g. "ACTION_VIEW" specifies the component is eligible to render contents to users. Data field specifies the type of data to operate on. Normally, data is identified by URI (Uniform Resource Identifier). Category field gives additional information about the action to execute.

A public intent receiver may request the senders to have permissions sometimes.

3.3 Threat model

As illustrated in Figure 3.1, the upper two scenarios represent potential implicit intent hijacking. Attackers typically need to first decompress Android installation package(apk) to extract AndroidManifest.xml. Secondly, attackers need to filter out publicly-accessible intent receivers that set the EXPORTED flag or includes at least one $\langle intent - filter \rangle$. Thirdly, attackers need to declare a component with the same $\langle intent - filter \rangle$ in his malicious app. When the malicious app is installed on device, the malicious component will have the potential to hijack the implicit intent that matches the declared $\langle intent - filter \rangle$. If the intent receiver is a service, the

attack will lead to Denial-of-Service(DoS) attack. If the intent receiver is an ordered broadcast receiver, the attack can lead to DoS, or even Man-in-the-Middle(MITM) attack. The third scenario indicates the inappropriate processing of data extracted from an incoming intent. E.g., if the intent receiver does not check the validity of extracted data before operating on the data, exceptions may be triggered at runtime and crash the whole app immediately.

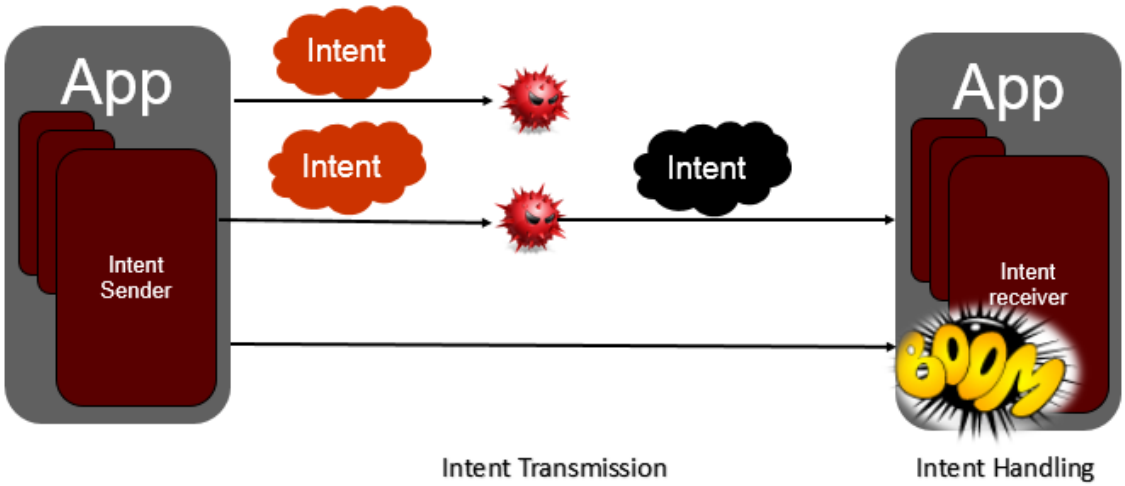

Figure 3.1: Overall threat model

4. STUDY ON ENSURING INTENT REACHABILITY TO TARGET RECEIVER

Intent Hijacking has been reported for quite a while, ever since 2011[2]. However, how it is triggered has not been sufficiently studied. To gain an in-depth understanding of intent hijacking prerequisite and complexity, we study the questions "What rule Android system follows to determine "the most appropriate" intent receiver?" and "How easily can intent hijacking be launched?" by designing a series of experiments for each type of intent receiver. In this section, we will try to answer the question and present what we can do about the problem from Android systems' perspectives.

4.1 Experiments

4.1.1 Activity experiments

We have system default browser and Chrome browser installed in our device. Both browsers are publicly-accessible and can handle an implicit intent as below:

```
Intent i = new Intent();
i.setAction(Intent.ACTION_VIEW);
i.setData(Uri.parse("https://howdy.tamu.edu"));
```

When Android systems resolve the implicit intent, they will search for an activity that can render the webpage at "https://howdy.tamu.edu" to users. Since both system default browser and Chrome browser can deal with the intent, the user will be prompted to choose which browser the intent should go to if a default choice has not already been set, as shown in Figure 4.1. Activity hijacking is difficult to be successful because Android system always prompts a dialog explicitly to users. This

kind of hijacking cannot succeed without user's notice. An attacker has to struggle to fool users with a confusing app name and implement a malicious activity that is similar enough to the original acitivity so that users cannot differentiate them at all.

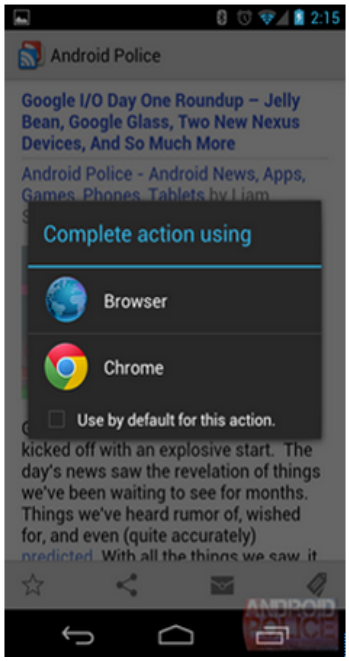

Figure 4.1: Activity experiments result

4.1.2 Service experiments

We have two apps, which contain Service A and B respectively, installed in our device. Both Service A and B are publicly-accessible and can handle the exactly same implicit intent (same action, data, category fields, etc. in $\langle intent - filter \rangle$): After sending an implicit intent to Android system, unlike Activity Experiment, there is no prompt dialog shown at all. One of the matching service will be started automatically. Instead of starting a matching service randomly as mentioned in

previous work, in fact, Android system will start service deterministically. In our experiment, we make one service to be loaded to Android system firstly. As for the other one, we load it secondly but start it immediately after loaded. As shown in Table 4.1, Android system always starts the first loaded service to respond to the implicit intent. Service hijacking is not apparent to users because there is no user interface involed explicitly. Therefore, service hijacking is not hard to be launched successfully by attacker.

	Service A	Service B	Result
Experiment I	Load Firstly	Start Firstly	A starts
Experiment II	Start Firstly	Load Firstly	B starts

Table 4.1: Service experiments result

4.1.3 Ordered broadcast receiver experiments

Similarly, Ordered Broadcast receiver I and II can handle the same implicit intent. The only difference with previous experiments is the involvement of priority in broadcast receiver. Priority is an integer value greater than -1000 and less than 1000. For two broadcast receivers with different priority, the one with larger value is always the first one to receive ordered broadcast firstly. A question is, "which one of receiver I and II will be the first to receive ordered broadcast when they have the same priority?" After the following three experiments in Table 4.2, we found loading-sequence dominates when two broadcast receivers are equally eligible to receive an ordered broadcast.

	Receiver I	Receiver II	Result
Experiment I	P:300	P:200	R1→R2
Experiment II	P:300 (loaded firstly)	P:300 (start firstly)	R1→R2
Experiment III	P:300 (start firstly)	P:300 (loaded firstly)	R2

Table 4.2: BroadcastReceiver experiments result

4.1.4 Experiment summary

From the three series of experiments above, we draw the conclusion that, (i) Android system will follow "First Loaded, First Served" rule when more than one services/order broadcast receivers can deal with the same implicit intent. (ii) Service/Ordered Broadcast Receiver hijacking is easier to be successful because users will not be provided the chance to select their targets explicitly.

4.2 Load sequence

Since Android system relies on "First Loaded, First Served" rule to select the most appropriate implicit intent receiver, we studied the factors that influence app loading sequence. First of all, where to put an app installation package plays a significant role. At device boot time, apps will be loaded one by one. According to AOSP, Android system will scan the following directories, if exists, for application installation packages in sequence.

```
vendorOverlayDir:        /vendor/overlay/
framworkDir:             /system/framework/
privilegedAppDir:        /system/priv-app/
systemAppDir:            /system/app/
vendorAppDir:            /vendor/app/
```

```
oemAppDir:                  /oem/app/
mAppInstallDir:             /data/app/
mDrmAppPrivateInstallDir:   /data/app-private/
```

According to my experience with LG nexus 5 with native Android OS and Xiaomi Phone with customized MIUI OS, systemAppDir and mAppInstallDir are the most commonly-used directories for app storage. For those apps that come with device are stored in "/system/app/" directory, such as browser.apk, calendar.apk, etc. User-installed apps, such as skype.apk, facebook.apk, etc., are stored in "/data/app/" directory. Root privilege is needed to access most of the directories listed above. From an attacker's point of view, it is very difficult to obtain root privilege of device. So it is impossible for a malicious app to place itself in vendorOverlayDir. What attackers can manipulate is the fact that apps in each directory are sorted according to the alphabetical order of their file name. In other words, if a malicious app, which contains the malicious intent receiver for hijacking, is named "A.apk" and put in directory"/data/app", it is likely to deny the benign services or become the first one to receive ordered broadcast when implicit intent is resolved by Android systems.

4.3 System architecture

The prerequisite of the threats is the presence of malicious service/broadcast receiver in Android devices. Therefore, we propose our ***first defense strategy***: Prevent malicious services/broadcast receivers from being installed to devices, as shown in Figure 4.2 - Approach 1. Existing work focuses on statically analyzing decompiled dalvik executables and looking for the sending of implicit intent without permission protection. We approach the problem in a more lightweighted way. As mentioned earlier in the section, malicious components need to declare exactly the same $\langle intent - filter \rangle$ information as the benign component in order to hijack the

implicit intent for benign component. For developer-defined actions, the best practice is to use package name as a prefix to ensure uniqueness according to Android Developer Guide[12]. However, the best practice is not followed by a lot of developers because of its strictness. But normally the action field in an $\langle intent - filter \rangle$, if as a developer-defined action, is always related to the app's package name. Therefore, we can determine whether to reject the installation of an app based on the following rule: **If any intent receiver in an app declares any developer-defined action that is completely irrelevant to the apps package name, the app will be rejected.** The checking procedure is completed by our instrumented Android-framework layer module. Since the original flow to install an app needs to decompress and extract information from AndroidManifest.xml anyway, in our change, the only extra step is to extract the action field of an $\langle intent - filter \rangle$ and send to our checking module. The time overhead is nearly negligible. Our ***second defense strategy*** is to provide selection dialog with matching intent receivers as options, as shown in Figure 4.2 - Approach 2. This strategy is inspired by Android system's strategy to resolve multi-activity matching problem. We change the control flow of how an implicit intent is processed and inject code to show a selection dialog when there are more than one components can fulfill the implicit intent's request. Users differentiate components by their package names.

4.4 Defense evaluation

Defense at Installation is effective at reducing the vulnerability, but the strictness may lead to false positives. We checked the top 50 highly-downloaded apps in Google Play Store in order to understand how many developers are accustomed to using package name as prefix when defining implicit intent actions. There are 335 publicly-accessible intent receivers, 670 actions (482 Google-defined actions and 188 developer-

Figure 4.2: Defense architecture

defined actions) in the 50 apps. We found only 59 of 188 follow the best practice. We studied developers' package naming habits and found out nearly all developers put their companies' names immediately after "com", "net", etc. Therefore, we set our reject rule to require "a developer-defined action should at least contain company name as a part". With this change, FP rate is reduced by 44% further. In our implementation, when an app does not satisfy the rule, we show users a warning that the app is suspicious. Users are given an option to continue the installation if they trust the app. Another improvement to reduce FP rate is we only applies the rule to apps from untrusted sources. We trust apps downloaded from popular online markets with strict malicious review mechanism.

Defense at Runtime depends on users explicit selection to prevent the vulnerability. Although there may be chances for a malicious app to be installed as well, we think the probability should be lower than before.

Our defense strategies are completely from Android System's perspective and does not lead to extra burden for device users.

5. DESIGN AND IMPLEMENTATION OF FUZZINGDROID

This section focuses on how to expose the deficiency of intent handling code in an intent receiver before an app is released to market. Our idea is to fuzz the publicly-accessible intent receiver using random and unexpected intents and monitor debugger console for triggered exceptions. We designed and implemented a tool called FuzzingDroid to assist developers in testing their apps. FuzzingDroid consists of two parts. One is running on device as an Android app in the same environment with other apps to be tested. This part is responsible for sending intents to app components to be diagnosed. The other part runs as an Android framework-layer module, which takes charge of generating different extras field for a fuzzing intent. We will discuss the details in the rest of this section.

5.1 FuzzingDroid design

The overview of FuzzingDroid is depicted in Figure 5.1. For each target app, the fuzzing workflow consists of the following 5 steps:

(1) Extract publicly-accessible intent receivers from AndroidManifest.xml

(2) Generate intent prototype with data, action, categories and flags

(3) Send intent prototype to Android system

(4) Intercept extras extraction call and Return fuzzing extras values

(5) Monitor exceptions

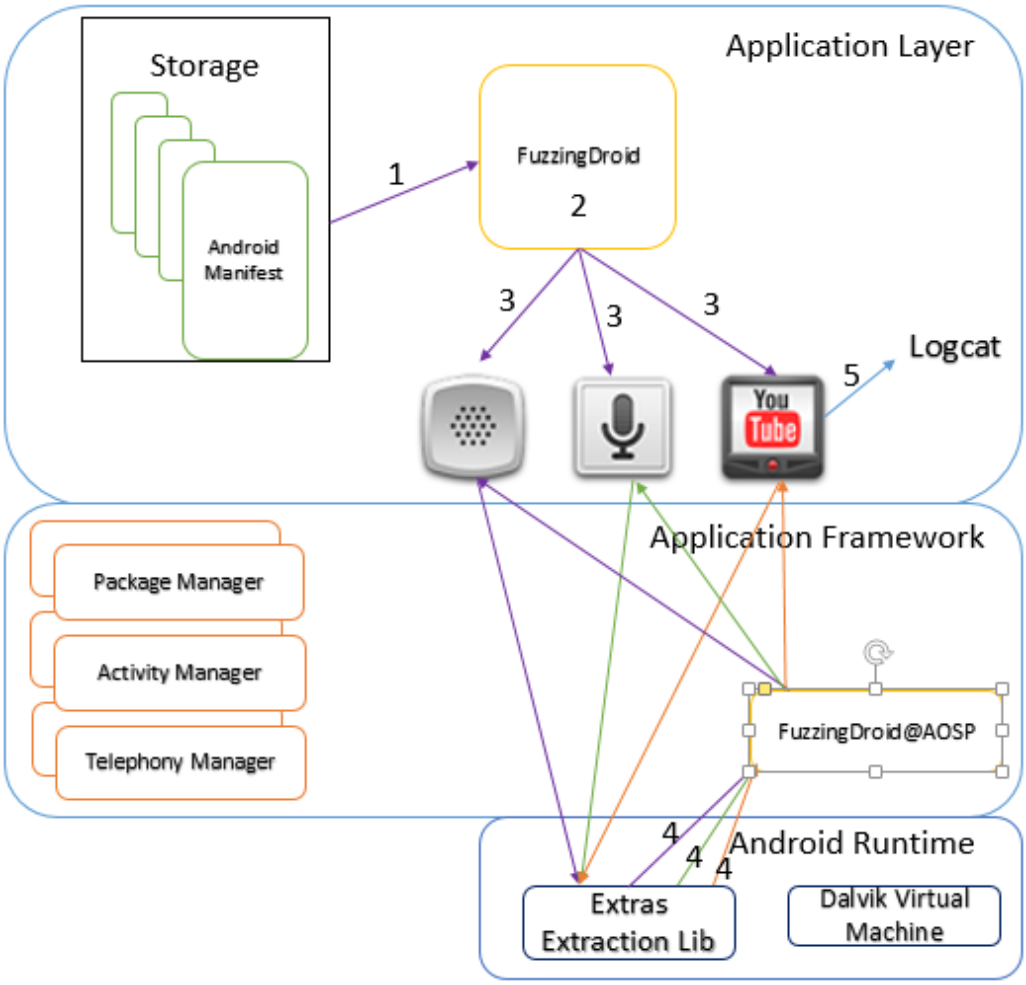

Figure 5.1: FuzzingDroid architecture

5.1.1 Extraction of publicly-accessible intent receiver

Since each publicly-accessible intent receiver is described in AndroidManifest.xml and cannot be altered after installation, it is straightforward to parse this file for fuzzing targets. We care about package name, $\langle intent - filter \rangle$ information, such as $\langle action \rangle$, $\langle category \rangle$, $\langle data \rangle$, $\langle flag \rangle$, etc. and the permissions that protect the receivers. Package name is an app's unique identifier. It helps tracking the fuzzing process when FuzzingDroid is used to test apps on a large scale. $\langle intent - filter \rangle$

tag indicates the detailed information about what an implicit intent should be in order to be received by the its target component. As mentioned earlier, we only care about those intent receivers that set EXPORTED flag or contain at least one $\langle intent - filter \rangle$ tag because only those intent receivers can receive implicit intent. To increase test coverage, our FuzzingDroid is granted commonly-used normal and dangerous permissions in order to be capable of fuzzing those intent receivers that are protected by these common permissions. For intent receivers that are protected by signature or higher permissions or any permissions our FuzzingDroid does not have, the intent receiver will not be tested further.

5.1.2 Generate intent prototype with data, action, categories and flags

In the second step, FuzzingDroid will construct intent prototypes with information collected from $\langle intent - filter \rangle$ in the first step. Intent prototype is semifinished intent without "Extras" field. With Action, Category, Data and Flag fields, the intent prototype should be able to reach its target receiver successfully. For each publiclly-accessible intent receiver, there will be at least one intent prototype.

Set Action: It is possible for an intent receiver to declare more than one action. But an intent object can only set one specific action. To fuzz an intent receiver efficiently, we will start as many threads as declared actions in an intent receiver. Each thread sends its own intent prototype with a unique action.

Set Category: It is possible for an intent receiver to declare multiple categories. An intent prototype must contain all the declared categories to be matched by its target receiver.

Set Data: We can infer data type that component is able to handle from $\langle data \rangle$ tag inside $\langle intent - filter \rangle$. Data is specified by scheme, host, port and path for each part of the URI scheme://host:port/path. We store common data types as URI in

FuzzingDroid. Our FuzzingDroid supports image/*, video/*, http/https. When no $\langle data \rangle$ tag in $\langle intent - filter \rangle$ or required data is unsupported by FuzzingDroid, our fuzzing intent will go without any Data URI.

Set Flag: Flags are predefined values for instructing Android system how to launch an activity. Our FuzzingDroid sets flag as "FLAG_ACTIVITY_NEW_TASK" when receiver is activity.

5.1.3 Send intent prototype to Android system

Our FuzzingDroid sends the intent prototype to Android system. With all the important fields properly filled in in last step, the intent prototype should be matched by its intent receiver, e.g. FuzzingDroid sent an implicit intent prototype for Chrome, Chrome should be shown as an option in the selection dialog after the implicit intent is resolved by Android system. The way an intent is sent to Android system depends on the type of the intent receiver. If intent receiver is Activity, "startActivity(Intent i)" or "startActivityForResult(Intenti, int requestCode)" will be used; If intent receiver is Service, "startService(Intent i)" will be used; If intent receiver is broadcastReceiver, "sendBroadcast(Intent i, String rcvrPermission)" or "sendOrderedBroadcast(Intent i, String recvrPermission)" will be used.

5.1.4 Intercept extras extraction call and return fuzzing extras values

The extras field of an intent is the core fuzzing variant part. Extras are key-value pairs that provide extended information to the intent receiver. Keys are strings while values can be any Java primitive type or class. E.g. If I have an intent to send an email message, I can also include extra pieces of data here to supply a subject, body, etc. FuzzingDroid sends intent prototypes without extras. When an intent receiver tries to extract extras value with key from intent prototype, it will use Extras extraction library in Android Runtime Layer, as shown in Figure 5.1. since intent pro-

totype does not contain Extras field, the extraction call is intercepted and redirected to our FuzzingDroid@AOSP module, which is in Android framework-layer. FuzzingDroid@AOSP stores our fuzzing variants for supported Java primitives, composites and objects. Upon receiving a request, FuzzingDroid@AOSP will return a random value of proper type to the intent receiver for further processing. E.g. An intent receiver may extract a string password according to a string username with "passwd = intent.getStringExtra(username)", Our injected code will redirect "getStringExtra()" to FuzzingDroid@AOSP, which may return empty string, or null string, or string with any combination of punctuation, digits and unreadable characters randomly. These returned value will be assigned to "passwd" for further processing in the intent receiver being fuzzed.

5.1.5 Monitor exceptions

Logcat, which is a console debugger for collecting and viewing system debug output, is monitored for abnormality. Logcat output can be filtered based on the package name nad emergency of information, e.g., Error, Warning, Info, etc.. During fuzzing process, we restrict logcat to only output error information from a specific package name that is being fuzzed at that time because we only care about the uncaught exceptions, which will lead to errors, triggered by our FuzzingDroid. A example of Logcat output is shown in Figure 5.2.

5.2 FuzzingDroid implementation

5.2.1 Architecture

We built our FuzzingDroid as an app on Android system for three reasons. Firstly, it is convenient to monitor fuzzing progress. FuzzingDroid is designed to output info about which intent receiver is being fuzzed, what are the fuzzing parameters, etc. Secondly, it is convenient to fuzz intent receivers protected by normal permissions

```
03-26 11:56:08.128: E/AndroidRuntime(593): Caused by:
java.lang.NullPointerException
03-26 11:56:08.128: E/AndroidRuntime(593):  at
gaurav.android.CalcActivity.reset(CalcActivity.java:341)
03-26 11:56:08.128: E/AndroidRuntime(593):  at
gaurav.android.CalcActivity.onCreate(CalcActivity.java:58)
03-26 11:56:08.128: E/AndroidRuntime(593):  at
android.app.Activity.performCreate(Activity.java:4465)
```

Figure 5.2: Logcat sample error output

by granting our FuzzingDroid with those permissions. In other words, more intent receivers can be fuzzed. Thirdly, ease of use. Fuzzing process gets ready simply after develpers download our FuzzingDroid on their devices.

In case of crashing system with too much fuzzing intent in a short time, we set a 5-second buffer time before our FuzzingDroid switching to fuzz another intent receiver. Each intent prototype is is used 5 times for fuzzing and taken care by one thread at the same time. Developers can easily adjust parameters to accelerate FuzzingDroid according to their devices capability.

FuzzingDroid can be used to test multiple apps synchronously. Test in batch is a time-consuming task. Therefore, we implement the core of our FuzzingDroid as a background service in Android systems. To facilitate the fuzzing process, FuzzingDroid can be configured to fuzz multiple receivers in an app synchronously, or even fuzz multiple apps at the same time. For each intent receiver in each app, FuzzingDroid will dispatch as many threads as the number of action fields declared in the intent receiver to fuzz simultaneously. Each thread will send 5-10 fuzzing intent prototypes to its target intent receiver for coverage.

We use proclogcat[13], which is a perl script designed to track a specifc packages logcat messages in a stable and reliable way, to monitor fuzzing targets abnormalities.

Usage of proclogcat:

```
adb logcat | ./proclogcat [package_name]
```

5.2.2 System modification

We modified APIs that can extract extras value in AOSP. There are two ways for an intent to carry extras. The first one is storing extras directly in intent and read them with intent.getXXXExtra(String key). The second one is storing extras in a mapping object called Bundle and let intent carry Bundle. Extras can be read with Bundle.getXXX(String key). XXX can be Int, IntArray, IntegerArrayList, etc. in both cases.

When these APIs are called by an intent receiver, the call will be redirected to FuzzingDroid@AOSP, which manages all the random variants of different data types. It will return the variants to the intent receiver for further processing. With this mechanism, we never need to worry about the Extras key collection challenge mentioned in related works.

5.3 FuzzingDroid evaluation

FuzzingDroid satisfies the following three basic usefulness criteria:

(1) Robust: FuzzingDroid can handle complicated real-world apps and seldom crashes.

(2) Black-box: FuzzingDroid does not need app source code. It simply needs AndroidManifest.xml, which can be obtained by decompressing application package, to determine fuzzing targets.

(3) Automated: FuzzingDroid needs very little human effort assuming apps have already been installed on devices.

Usefulness is just baseline, we evaluated FuzzingDroid performance further from two perspectives, execution overhead and bug detection.

We evaluated FuzzingDroid with ***Device:*** LG Nexus 5 with 2GB RAM memory and

Qualcomm MSM 8974 2300.0 MHz (4-core) CPU. ***Dataset:*** 47 highly-downloaded apps from Google Store, 46 popular apps from other app online markets, 45 core system apps from Nexus 5 and 32 core system apps from Xiaomi 2s phone. These apps cover nearly every possible categories. Detailed distribution figures are shown in Figure 5.3 and 5.4.

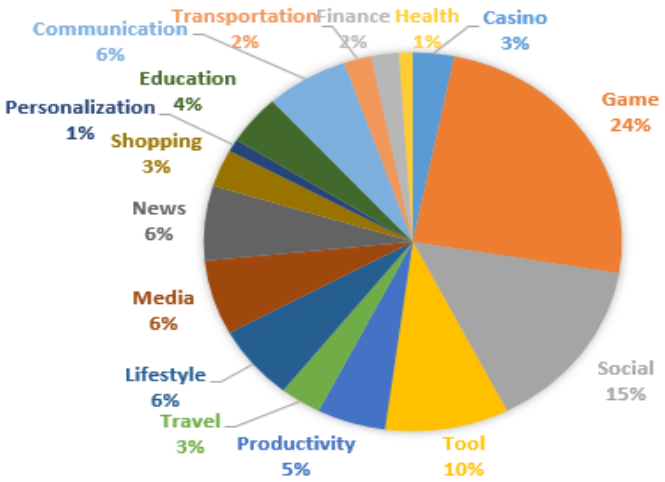

Figure 5.3: Distribution of user-download apps by category

5.3.1 Execution overhead

Considering the limitation of device, I specified a 5-second buffer time between the fuzzing of any two different intent receivers. The average time to fuzz an app is less than 1 minutes. It took about 2 minutes if taking fuzzing preparation(E.g. Determining fuzzing target, Setting log filter, etc.) into consideration.

We used an app called "CPU Memory Monitor" to evaluate the CPU and memory utilization during fuzzing process. "CPU Memory Monitor" is a highly-downloaded system tool with excellent reviews from its users.

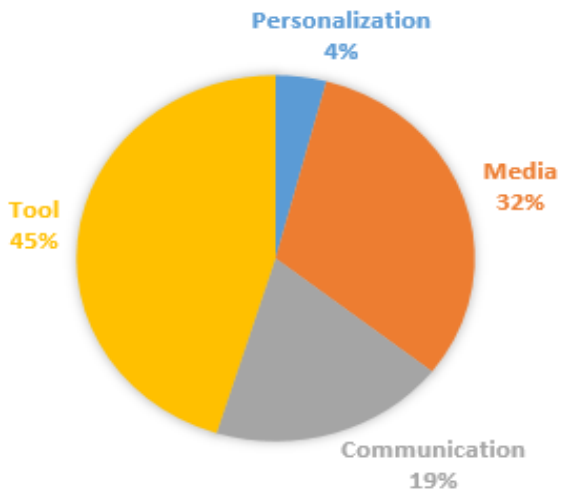

Figure 5.4: Distribution of system apps by category

Figure 5.5 and 5.6 shows the CPU and memory utilization in device respectively over a 80-second period, which is a time slot our FuzzingDroid works to test 8 Xiaomi system apps. During the period, there is a 5% increase in CPU utilization and 24MB increase in memory usage.

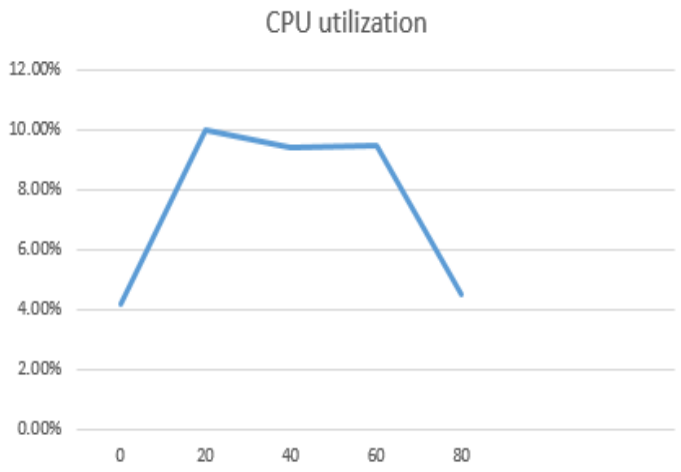

Figure 5.5: CPU utilization during a 80-second fuzzing period

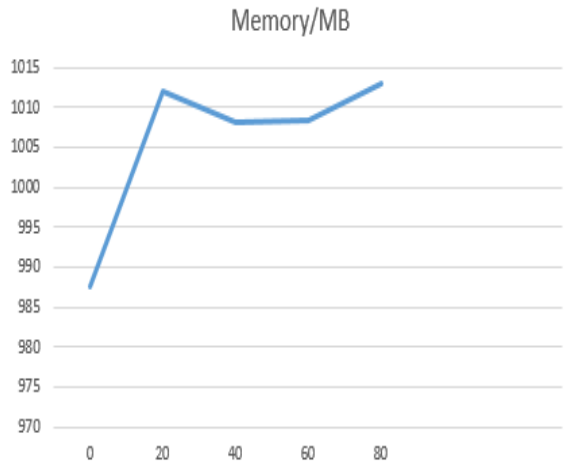

Figure 5.6: Memory utilization during a 80-second fuzzing period

5.3.2 Bug detection

Our FuzzingDroid successfully crashes 49 of 170 apps in total (31 crashes from user-download apps from online app markets, 7 crashes from Nexus 5 system apps, 11 crashes from Xiaomi phone). Nexus 5's system version is Android 4.4 KitKat. All system apps in Nexus 5 are developed and maintained by Google inc. Xiaomi phone's system version is MIUI, which is a customized Android system. Xiaomi OS development team keeps several system apps from Google as the system apps in MIUI, such as Calendar, Bluetooth, etc. Besides, Xiaomi added a lot of their apps as system apps in MIUI as well, such as MiTalk, MiuiVideo, etc. Since our test device is Nexus 5, we have to install Xiaomi's system apps as well. Some of the apps cannot be installed because of incompatibility issues. But we still got 32 core MIUI system apps installed. Although FuzzingDroid tested a smaller number of system apps from MIUI than Nexus 5, more crashes are triggered. Consider the overlapping issues, MIUI seems to have more intent handling deficiencies than Nexus 5.

Crashes are mostly caused by inappropriate handling of exceptions in program. For example, We found 22 NullPointerException, 7 ClassNotFoundException, 3 IllegalArgumentException, 4 NoClassDefinitionException and a bunch of other exceptions. 49 of all the triggered exceptions crashed test apps directly while 2 led to "Application Not Responding". When we were collecting the data, we only relied on the error messages from Logcat. There were also a number of exception warnings from Logcat output, however, those exceptions are handled well by developers and were not counted in our bug detection evaluation.

6. CASE STUDY

6.1 FuzzingDroid case study

Besides the bugs that crash apps, we also care about capability leaks during fuzzing process. Simply speaking, capability leak indicates the situation where the permission of an app may be used by other apps indirectly.

6.1.1 CleanMaster

Clean Master can get memory freed by killing background processes. This functionality should have been protected by "android.permission.RESTART_PACKAGES". Any app can invoke the capability-leaking component "com.cleanmaster.appwidget.WidgetService" using an intent with action as "com.cleanmaster.appwidget.ACTION_FASTCLEAN" to kill background processes.

6.1.2 FakeCall

Fake Call can fabricate incoming calls and short messages. Any app can reproduce an incoming call anytime by sending an intent with an "id" field to component "com.popularapp.fakecall.incall.CallAlarm". However, this functionality should only be invoked by apps with "WRITE_CALL_LOG" permission.

6.2 Case study for intent interception and manipulation

When playing a video with Xiaomi's system video player, user can capture a screenshot and save it to gallery by sending an intent with action "android.intent.action.CAPTURE_SCREENSHOT" as ordered broadcast. An integer field "capture_delay" in the intent extras will control the delay between the receiving and execution of taking screenshot. Malicious broadcast receiver can intercept the intent and modify the value of "capture_delay" field. The modified intent

can be sent to its original target (benign broadcast receiver) for further processing.

7. CONCLUSION

We conducted a robustness study on the life cycle of intent in Android Systems. From Android system's perspective, we implemented several framework-layer modules in Android Open Source Project to facilitate the reachability of implicit intents to their target receivers; From a developer's perspective, we implemented FuzzingDroid to test the robustness of intent handling by publicly-accessible intent receivers. We successfully triggered and detected logic flaws in several popular apps in Android markets. We evaluated our work from various aspects to prove its effectiveness.

As for future works, we will keep improving FuzzingDroid to fuzz more smartly with more representative fuzzing variants. Besides, how to expose more vulnerabilities other than bugs is a challenging and interesting problem we want to study in near future as well.

REFERENCES

[1] L. Davi, A. Dmitrienko, A. Sadeghi and M. Winandy, "Privilege Escalation Attacks on Android", *in Proceedings of the 13th International Conference on Information Security*, 2011, pages 346-360.

[2] E. Chin, A. P. Felt, K. Greenwood and D. Wagner, "Analyzing Inter-application Communication in Android", *in Proceedings of the 9th Annual Symposium on Network and Distributed System Security, MobiSys*, 2011, pages 239-252.

[3] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee and Guofei Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities", *in ACM CCS*, 2012, pages 229-240.

[4] Aravind MacHiry, Rohan Tahiliani and Mayur Naik, "Dynodroid: An Input Generation System for Android Apps", *in Proceedings of 9th Joint Meeting of the European Software Engineering Conference and the ACMSIGSOFT Symposium on the Foundations of Software Engineering* (Saint Petersburg, Russia, August 18-26, 2013), 2013, pages 224-234.

[5] Intent Fuzzer, `https://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx`, accessed Nov. 30th, 2014.

[6] Amiya Kumar Maji, Fahad A. Arshad, Saurabh Bagchi and Jan S. Rellermeyer, "An Empirical Study of the Robustness of Inter-component Communication in Android", *in Proc. DSN*, 2012, pages 112.

[7] Raimondas Sasnauskas and John Regehr, "Intent Fuzzer: Crafting Intents of Death", *in Proc. WODA+PERTEA 2014*, 2014, pages 1-5.

[8] Hui Ye, Shaoyin Cheng, Lanbo Zhang and Fan Jiang, "Droidfuzzer: Fuzzing the Android Apps with Intent-filter Tag", *in Proc. MoMM*, 2013, pages 6874.

[9] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujie Zhou and Haixin Duan, "Intent-Fuzzer: Detecting Capability Leaks of Android Applications", *in Proc. ASIA CCS14*, 2014, pages 531-536.

[10] Intent, `http://developer.android.com/reference/android/content/Intent.html`, accessed Dec. 1st, 2014

[11] Intent-filter, `http://developer.android.com/guide/topics/manifest/intent-filter-element.html`, accessed Dec. 1st, 2014

[12] Action tag, `http://developer.android.com/guide/topics/manifest/action-element.html`, accessed Dec. 2nd, 2014

[13] Proclogcat, `https://github.com/jasta/android-dev-tools/blob/master/proclogcat`, accessed March. 15th, 2015