An Empirical Study of the Robustness of Inter-component Communication in Android

Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi
Purdue University
West Lafayette, IN, USA
{*amaji, faarshad, sbagchi*}*@purdue.edu*

Jan S. Rellermeyer
IBM Research
Austin, TX, USA
rellermeyer@us.ibm.com

***Abstract*—Over the last three years, Android has established itself as the largest-selling operating system for smartphones. It boasts of a Linux-based robust kernel, a modular framework with multiple components in each application, and a security-conscious design where each application is isolated in its own virtual machine. However, all of these desirable properties would be rendered ineffectual if an application were to deliver erroneous messages to targeted applications and thus cause the target to behave incorrectly. In this paper, we present an empirical evaluation of the robustness of Inter-component Communication (ICC) in Android through fuzz testing methodology, whereby, parameters of the inter-component communication are changed to various incorrect values. We show that not only exception handling is a rarity in Android applications, but also it is possible to crash the Android runtime from unprivileged user processes. Based on our observations, we highlight some of the critical design issues in Android ICC and suggest solutions to alleviate these problems.**

***Keywords*-android, fuzz, security, smartphone, robustness, exception**

I. Introduction

As of December 7, 2011, a lot of incidents related to smartphones have appeared as headlines in the media over the past two weeks. A Youtube video posted by a security researcher received more than 1.5 million views after he exposed a contentious logging program in a "reputed" network intelligence program [1] for smartphones. An iPhone exploded on board an Aussie flight causing temporary panic among the passengers and the crew [2]. In another part of the world, the authors of this paper had their fair share of extraordinary experiences as well. One of the authors of this paper found his newly purchased smartphone magically bypass the screen lock after pressing the power key twice in succession [3]. The list does not end here as the authors had to pull out the batteries from their experimental phones time and again to un-"freeze" them! How robust are smartphones of today? This is the question we answer in this paper. Specifically, we evaluate how robust are Android's built-in and best-seller applications to malformed Inter-component Communication messages.[1]

We selected Android as the mobile platform for our study for obvious reasons: it has the leading market share in smartphones and its codebase is open. In three years since its release, Android has become the leading smartphone OS in the world with a staggering sales figure of 60 million phones in the third quarter of 2011 alone [4]. A software with such large customer base needs to be very robust and secure, otherwise even minute defects may overshadow its myriad desirable features. Android has also received significant attention from research and developer communities. Its modular approach to application development allows mutually untrusting applications to share their functionality. To protect these applications from one-another, Android assigns different user IDs (UID) to each application and runs them in isolated virtual machines. However, in a collaborative environment, applications need to share data which is supported by Android with a flexible communication mechanism. Communication, traditionally, introduces new vulnerabilities and exposes applications to a variety of stressful conditions, a classic example being noisy data from sensor equipment. Unexpected input also has the potential to break the security measures employed by a system and expose sensitive data. In case of smartphones, sources of inputs can be significantly diverse—these include touchscreen, keyboard, radio, microphone, sensors, untrusted third-party applications, or data from one of many network drivers—and therefore *it has great potential for receiving unexpected data*. Given the unorthodox techniques people employ to bypass password locks on their smartphones [5], receipt of *unexpected* data is not a rarity. Our objective in this paper is to see how well Android reacts to unexpected data, and more specifically to test its Inter Process Communication primitives. We define robustness as the ability to handle unexpected data gracefully, therefore, lack of robustness would imply an application crashing in response to an IPC message. In the context of Android applications, these crashes manifest as uncaught exceptions in the stack trace.

Inter Process Communication in Android takes place in one of two ways—Binders, where an application creates a proxy for a remote object (having known interface) and can invoke remote methods, and Intents, a data container which is passed from one application component to another through mediation of the Android Runtime. Of these, Intents allow dynamic target selection and runtime binding, i.e., the sender of an Intent does not need to know anything about the receiver. Due to its dynamic nature, Intents have a flexible structure. It is easy to generate Intents, and therefore, can become a simple tool for an adversary who wants to compromise a system. For all these reasons, we generated

[1]In this paper, we use the term Inter-component Communication to cover both intra and inter-application messages.

random and semi-valid Intents and tested how Android reacts to these. Traditionally, researchers have used fuzz testing for testing the robustness of software systems. In fuzz testing, random input is fed to an application, e.g., sending random parameter values to the system calls. Fuzz testing has been used with considerable success to evaluate the robustness of various operating systems [6], [7], [8], [9]. More intelligent test case generation for robustness testing can be seen in [10]. However, such evaluation of mobile OSes is rare in the research literature.

Our objectives in this study are threefold—to test how robust Intent handling is, to discover vulnerabilities through random (or crafted) Intents, and to suggest recommendations for hardening of Android IPC.

With these goals, we developed our Android robustness testing tool, JarJarBinks (in remembrance of the Gungan warrior of Star Wars fame, whose unusual accent created significant problems for the Droid). JarJarBinks includes four Intent generation modules—semi-valid, blank, random, and random with extras, and the ability to automatically send a large number of Intents to all the components. JarJarBinks runs as a user level process, it does not require knowledge of source codes of the tested components, and can be easily configured for the robustness testing on any Android device. During our experiments we sent more than 6 million Intents to 800+ application components across 3 versions of Android (2.2, 2.3.4, and 4.0) and discovered a significant number of input validation errors. In general less than 10% of the components tested crashed; all crashes are caused by unhandled exceptions. Our results suggest that Android has a sizable number of components with unhandled `NullPointerExceptions` across all versions. Though Android's exception handling capability has improved significantly since v2.2, its latest version (4.0) displays a larger number of environment-dependent failures. These failures do not happen predictably in time and are therefore insidious from the point of view of testing.

The most striking finding that we have is the ability to run privileged processes from user level applications without requiring the user-level application to be granted any special permission at install time. We found three instances, where we could crash the Android runtime from JarJarBinks. Such a crash makes the Android device unusable till it is rebooted. This has huge potential for privilege escalation, denial-of-service, and may even lead to more security vulnerabilities, if an adversary could figure out how to have these malformed (or "fuzzed") Intents be sent out in response to some external message. To improve software design from the point of view of reliability, we found that subtyping combined with Java annotations can be used very effectively to restrict the format and content of an Intent. Through this mechanism, the attack surface of Android can be reduced significantly.

The rest of our paper is organized as follows. We begin with an overview of Android and explain key terminology in Section II. Section III presents the design of JarJarBinks and explains our Intent generation methodology. The next Section presents results obtained from our experiments and suggests some guidelines for secure Android application development. Section V presents ICC design recommendations for securing Intents followed by discussion of Future Work.

II. Android Overview

A. Android Architecture

Android is an open source platform for mobile system development with a standard Linux operating system, a customized runtime, a comprehensive application framework and a set of user applications. Based on Linux kernel, it provides a robust driver model, security features, process management, memory management, networking assistance and drivers for a large set of devices. The runtime comprises of core libraries and Dalvik [11], a register based [12] virtual machine optimized to run under constrained memory and CPU requirements. Application framework provides developers APIs for building user applications (popularly called apps).

B. Android Application Components

Here we first explain the different kinds of application components in Android and then explain how the different components coordinate among themselves to achieve a task. This background would be essential to understand the experimental methodology that we have developed because we choose the inter-component messages (called Intents in Android) as the target of our fuzz testing. To understand how Android application components co-ordinate to achieve a task, consider two sample applications (Email and Contacts) shown in Figure 1, that co-operate in replying to an email. Consider, a user launching an email application from home screen. This starts an *Activity* (user interface (UI)) showing the user's *Inbox*. She then clicks on an email she wants to read which starts another UI showing a particular *Email* message. To reply, she clicks *Reply* button to invoke a third activity where she can type her response. Consider, she wants to copy her reply to more recipients, so she hits the *"cc"* button to find the address of the recipient. This invokes a fourth activity, i.e., *Select Contact* in Contacts application showing the available email addresses. This fourth activity to user appears as a part of email application but in reality it is from a separate application (Contacts) which runs in a separate process. Further, the main activity in Contacts application, i.e., *Select Contact* calls a *Content Provider*, another application component for data storage, to retrieve the recipient's email address. The sequence of called activities, *Inbox, Email, Reply, Select Contact* to achieve a given *task* involves inter-component communication which can be either inter-application or intra-application.

Each user application in Android (a *.apk file) typically runs in a separate process and can be composed

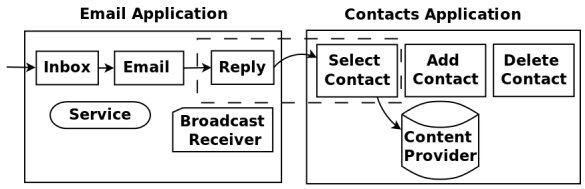

Figure 1. Android Application Components

of *Activities*, *Services*, *Content Providers* and *Broadcast Receivers*. These four components communicate through messages called *Intents* that are routed through Android runtime and the Kernel. The underlying runtime manages the Inter-component Communication. At application installation time, the contract with the runtime is specified in *AndroidManifest.xml*. This contract details on type of components, application permissions, etc. Here we briefly define each of the component types.

Activities: An Activity is a graphical component, which is used to provide the client with a user interface. It is invoked when a user launches an application. An activity can send and receive Intents to and from runtime. It is implemented by extending the *Activity* class while its life cycle is managed by a module in application framework layer called *Activity Manager*.

Services: A Service is used when an application task needs to run in background for a longer time period. For example, a user can run music player in background. Also, a component can bind to a Service to send a request, e.g., a music player Activity can bind to a music player Service to stop the current song that is being played.

Content Providers: A Content Provider is used to manage access to persistent data. The data can be shared between multiple Activities in different applications. Contacts application, as an example, can use the content provider to get a person's phone number.

Broadcast Receivers: A component that is solely responsible to receive and react to event notifications is called a Broadcast Receiver. For example, in SMS application, the Broadcast Receiver component receives an SMS message and displays an alert.

C. Android IPC

The inter-process communication (IPC) in Android occurs through a kernel space component called Binder (`/dev/binder`), a device driver using Linux shared memory to achieve IPC. The higher level user space components know how to use the binder, i.e., how to pass data represented by *Intents* to Binder. Specifically, when a given component, e.g. Activity Manager, wants to do IPC (either an IPC send or an IPC receive) at OS boundary, it opens the driver supplied by the Binder kernel module. This associates a file descriptor with the thread that called binder, and this association is used by the kernel module to identify the caller and callee of Binder IPCs. All IPC at OS boundary takes place through this descriptor. At the higher level, application-runtime boundary, the application components send Intent messages, e.g., an Activity sends Intents to Activity Manager.

D. Intents

Intent, a data container, is an abstraction for an action to be performed and forms the core of Android's IPC mechanism. An Intent encapsulates *action, data, component, category* and *extra* fields in its object. As an example, an action can be *dial*, with data as *phone number* and *component* as phone application's main activity. *Category* and *extra* fields give extra information on *action* and *data* respectively. An Intent message can be specifically (**Explicit Intent**) sent to a target component by naming it or it could be resolved by runtime to find a target component. When the target is not explicitly specified in Intent message (**Implicit Intent**), the Android runtime resolves the target component to be invoked by looking up the Intent message and matching it against components that can handle the Intent. A given target component can handle an Intent, if it is advertised in a tag called *Intent-filter* in *AndroidManifest.xml*. Different ways in which Intents are sent by application components are: (1). By launching an Activity using `startActivity(Intent)` type of methods; (2). By sending to Broadcast Receivers using `sendBroadcast(Intent)` type of methods; (3). By communicating with a service using `bindService(Intent, ServiceConnection, int)` type of methods; (4). By accessing data through Content Providers.

E. Android Security

Android provides two important security mechanisms that are different from traditional Unix systems, i.e., application sandboxing and permissions. Sandboxing means each Android application (*.apk) is given its own unique UID at install time that remains fixed throughout its lifetime. This is different from traditional desktop systems where a single user ID is shared among different processes. In Android, since two applications run as two different users, their code may not be run in the same process, thus requiring the need of IPC. Moreover, applications are also assigned separate directories where they can save persistent data. Applications can specify explicitly whether it will share its data with other applications in *AndroidManifest.xml*.

Application permissions is a Mandatory Access Control (MAC) mechanism for protecting application components and data. To use resources, an application requests permissions through *AndroidManifest.xml* file using the *uses-permission* tag at installation time. For example an application that needs to monitor incoming SMS messages would explicitly specify permission of "`android.permission.RECEIVE_SMS`". To protect

or share an application's own components, an application can define and specify a certain permission for a caller. This mechanism gives fine-grained control of different protected features of the device but fixes these permissions to install time as opposed to runtime.

III. Experimental Setup

Of the two Inter Component Communication (ICC) primitives in Android—Intent and Binder—we use Intent as the subject of our robustness study due to its flexibility. Intents are used for a variety of purposes in Android applications which include but are not limited to—starting a new activity, sending and receiving broadcast messages, receiving results from another activity, starting and stopping a service etc. To support these operations across a myriad applications from multiple vendors over many versions, Intent messages have a flexible structure and therein lies the potential for vulnerability. In a vulnerability analysis of Android IPC, Chin *et al.* [13] argued that it is easy to spoof, snoop, and target Intents to specific application components unless these are protected by explicit permissions, which is a rare occurrence. Our experimental results concur with this analysis and show that the attack surface can go even deeper (i.e. up to the framework layer or lower as shown in [14]). Due to these reasons we chose Intents as the primary focus of our study. In essence, we try to answer the following questions:
(A) How well does an Android component behave in the presence of a semi-valid or random Intent?
(B) How robust are Android's ICC primitives? Can the Android runtime contain exceptions within an application?
(C) How can we refine the implementation of Intents so that input validation can be improved?
To evaluate (A), we sent explicit Intents to each Activity, Service, and Broadcast Receiver registered in the system. We evaluate (B) by sending a set of implicit Intents and answer (C) by presenting a qualitative assessment in Section V.

A. Testing Tool

We built our robustness testing tool, JarJarBinks, from Intent Fuzzer at [15]. The initial codebase contained basic functions like displaying set of components registered in the system, and sending blank Intent messages to Broadcast Receivers, and Services. However, it did not support testing Activities. We added this key feature in JarJarBinks along with an Intent generation module described in Section III-B. Fig. 2 shows the location and operation of JarJarBinks (JJB) with reference to Android architecture [14]. It queries Android PackageManager to get a list of components (Activities, Services, and Broadcast Receivers) registered in the system and then uses ActivityManager to send Intents to these components. We use the following methods from Android API to send Intents: `startActivityForResult` for Activities, `startService` for Services, and `sendBroadcast` for Broadcast Receivers.

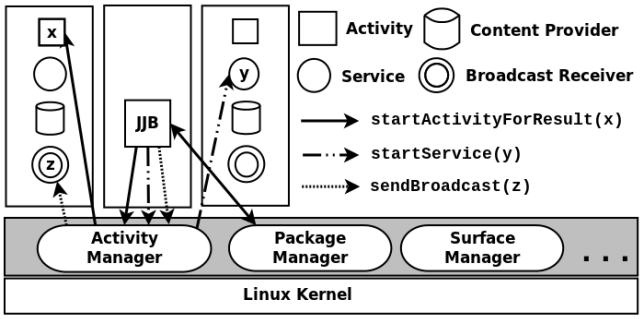

Figure 2. JarJarBinks: Interaction with Android Layers

One of the major challenges in automated testing of Android Activities is to close a callee Activity after sending an Intent. Typically, once a new Activity is displayed, it expects some interaction from the user and pauses the caller Activity. We resolved this by using `startActivityForResult()` and `finishActivity()` APIs in Android. Unlike `startActivity()`, `startActivityForResult()` can force-finish a child activity by using its `requestCode` as a handle. This way we could avoid manual intervention in most cases. Another design issue with automated testing of ICC in Android is to avoid resource exhaustion in the system (e.g., sending a continuous stream of Intents very fast would create a large number of Activities (windows) causing WindowManager to run out of resources). For this purpose, we used a pause of 100ms between sending of each successive Intent. This was sufficient to launch and finish a new Activity (or Service) in our testing environment. Though we did not explicitly test Content Providers in JarJarBinks, semi-valid content URIs were specified in some of our fault injection campaigns triggering parsing of these content URIs and corresponding permission checks.

It may be highlighted that one of our goals was to keep the implementation of JJB simple and less intrusive, thereby, not introducing new bugs in the firmware. We, instead, focus on a rigorous analysis of the results obtained from our experiments. Despite its simplicity, the volume and severity of failures generated through JJB is truly astonishing. One shortcoming of JJB is its semi-manual approach—our strategy of killing a child Activity (by calling `finishActivity`) did not work well in two situations: first, when a system alert was generated due to application crash, this could not be closed programmatically (we consider this as a good security design; JJB being a user-level application cannot hide system alerts), second, when an activity was started as a new task the caller could not close it by calling `finishActivity()` (this mostly happened while launching login screens of applications like Skype, Facebook, Settings etc.). Both these cases required manual intervention and will be addressed in our future work. In

the following section, we present an overview of our Intent generation module.

B. Generating Intents

An Intent message is essentially a data container having a set of optional fields—{Action, Data, Type, Package, Component, Flags, Categories, and Extras}—which can be specified by a caller. Of these, Action (an action to perform, e.g. to view or edit a contact) and Data (a URI for a data item, e.g. URI for a contact record on phone) are most frequently specified by a caller. Component specifies the target component, Flags control how an Intent is handled, Category specifies additional information about the action to execute, and Extras include a collection of name-value pairs to deliver more inputs to the target component. Type (content mime-type) is usually determined from Data (when it is specified), while, Package can be determined from Component if one is specified.

In JarJarBinks, we modify the fields Action, Data, Component, and Extras in a structured manner as part of a fault injection campaign and keep the other fields blank (we select Extras since this can potentially include random or malicious data from users). For most experiments Action is selected from a set of Android-defined action strings found at [16]. Generation of data URIs is a non-trivial operation due to the presence of a multitude of URI schemes. A URI consists of three parts URI := scheme/path?query, where scheme denotes URI type, path gives the location to the data, and query is an optional query string. At present we support the following URI schemes—"content://", "file://", "folder://", "directory://", "geo:", "google.streetview:", "http://", "https://", "mailto:", "ssh:", "tel:", and "voicemail:" in JarJarBinks. For each of these except "content://", we created a predefined set of semi-valid URIs. For "content://" URIs, JarJarBinks first queried the PackageManager to get a list of registered Content Providers in the system and then randomly selected one of them to build a content://provider URI. Our Intent generation can be broadly classified into two types.

1) Implicit Intents: Components in the system can advertise their ability to handle Intents by specifying Intent-filters in their manifest file. Implicit Intents do not specify a target, but are delivered to the best matching component in the system. The matching between sender and receiver is the responsibility of the Intent delivery mechanism of the platform. Intent-filters can restrict the Action of the Intent, the Category, or the Data (through both the URI and the data type fields) or any combination of the three. The test set for implicit Intents is therefore any Intent that matches at least one Intent-filter in the system. In order to generate Intents, we collect all Intent-filters of all applications and all restrictions of either the Action or the Category. On our target platform, we could not find components using the Data in Intent-filters. For each application and each of its Intent-filter, the following experiments were performed:

(A) *Valid Intent, unrestricted fields null:* We generate an Intent that matches exactly all the restricted attributes of the Intent-filter but leave all other fields blank. For example, if the Intent-filter specifies <action android:name="ACTION_EDIT" />, only this information is used to populate the Intent fields.

(B) *Semi-valid Intent:* We pick all Intent-filters that have at least one degree of freedom and set these fields sequentially to each of the valid literals we discovered in any other Intent-filter. For the above example, the Category field would be subject to fuzzing since only Action is restricted through the filter. Thus, the fuzzed fields are individually valid for some component in the system, but not their combination. Since each individual field in the generated Intent is valid, there is still a high chance that it is routed to *a* component.

2) Explicit Intents: Our goal here is to find how well the receiver of an Intent behaves after getting unexpected data. At a high level, our fuzz campaign on explicit Intents is distributed over three component types—Activities, Services, and Broadcast Receivers. For each component type, JarJarBinks first queries PackageManager to retrieve a list of components of that type in the system (e.g. all the Services, or Activities). After this, for each selected component (e.g. Calender Activity) JarJarBinks runs a set of four fuzz injection campaigns (FIC).

FIC A: *Semi-valid Action and Data:* Here a semi-valid Action string, and Data URI are generated as described earlier (refer Section III-B). However, the combination of the two may be invalid. For example, an Intent of this category may be Intent {act=ACTION_EDIT data=http://www.google.com cmp=com.android.someComponent}. During this FI, the Action and Data sets are combined to generate all known {Action, Data} pairs each generating a new Intent. Total number of Intents generated are |Action| × |Data| for each component. Fields other than Action and Data are kept blank.

FIC B: *Blank Action or Data:* In this experiment, we specified either Action OR Data in an Intent but not both together. Other fields are left blank. Intent {data=http://www.google.com cmp=com.android.someComponent} is an example of this FI. This campaign generates |Action| + |Data| Intents for each component.

FIC C: *Random Action or Data:* Here either Action OR Data is specified as described earlier, and the other is set to random bytes. An example of this type of Intent may be Intent {act=ACTION_EDIT data=a1b2c3d4 cmp=com.android.someComponent}.

FIC D: *Random Extras:* For this FI, we first created a set of 100 valid {`Action, Data`} pairs following Android documentation. For each of these pairs, 1-5 `Extra` fields were added randomly. The name of an `Extra` was selected from the set of Android defined `Extra` strings, while its value was set to random bytes. An example Intent can be shown as, `Intent {act=ACTION_DIAL Data=tel:123-456-7890 cmp=com.andro-id.someComponent has Extras}`.

Our choice of experiments is justified by the fact that an application component may get a malformed Intent either due to error propagation from other applications or from an active adversary. While FICs A and B verify the robustness of a callee component against null objects and incompatible actions, FICs C and D emulate the behavior of a potential adversary.

C. Machines and Firmware

We conducted our robustness test on three versions of Android, distributed on three phones and three computers—two of the phones (Motorola Droid) had Android 2.2 as its firmware (release date: June 2010 and nicknamed "Froyo"), while one (HTC Evo 3D) had Android 2.3.4 (release date: April 2011 and nicknamed "Gingerbread"); the computers all ran Emulators loaded with Android 4.0 in Linux environments (release date: October 2011 and nicknamed "Ice Cream Sandwich", the image of which was useful during long late night experiments with it). The HTC Evo was used for running experiments on implicit Intents. Experiments on explicit Intents, where we sent a large number (9000) of Intents to each Android component, being more time consuming, was run in parallel on two Droid phones (having identical hardware and firmware). The emulators were used for testing Android 4.0, the latest version of Android, for which a physical device has been available only in late November 2011, clearly not enough time for us to carry out experiments. Android 4.0 is a promising target of the study since it has been widely hailed as "the biggest Android update in ages" (PC Magazine) and is touted to bring real improvements to the Android platform. Initially, it was noted that the devices as well as the emulator had nearly 800 components (Activities, Services, and Broadcast Receivers combined) per version of Android which include a large number of third-party applications. In this paper, we focus our attention to Android framework and common applications that are pre-loaded into every Android distribution (e.g. contacts, calendar, messaging etc.). These application are also used by third-party application in implementing common functionalities. Hence, rigorous evaluation of these built-in applications are of prime importance. In Android namespace hierarchy, these applications all share the package name prefix of `com.android`. After filtering the list of components with this prefix we found 398 components (297 Activities, 42 Services, and 59 Broadcast Receivers) in Droid and 455 components (332 Activities, 54 Services, and and 69 Broadcast Receivers) in Emulator.

In addition to built-in applications, we also tested 5 Most popular (as on 3 Dec, 2011) free apps from Android Marketplace (recently renamed Google Play). These apps—Facebook, Pandora Radio, Voxer Walkie Talkie, Angry Birds, and Skype—had a total of 103 Activities and 11 Service components. Even though our set of Marketplace apps is small, the large number of Activities (103 as opposed to 294 in Droid) gives us a realistic comparison of their robustness with that of Android. Our experiments started by subjecting all these (Android and Marketplace) components to a flow of Intents from JarJarBinks over a seven day period. In the following section, we present our findings.

IV. Experimental Results

During the course of our experiments, more than 6 million Intents were sent to 800+ components across 3 versions of Android. We define an experiment as follows:
Choose one particular component and inject all the Intents targeted to that component. The injection is done according to the Fault Injection Campaigns (thus, if we are doing FIC A, the <`Action, Data`> pairs are changed to semi-valid values).
We collected execution logs from the mobile phones and emulators using `logcat`, a logging application in Android platform tools. This generated more than 3GB of log data which were later analyzed to gather information about the failures and their root causes. We define a **crash** to be a user visible failure, i.e., a system alert displaying the message `"Force Close"` (in Android 2.2) or `"Application x stopped unexpectedly"` (in Android 4.0). These failure messages manifest in the log files as a log entry stating `"FATAL EXCEPTION: main"` and are essentially effects of uncaught exceptions thrown by the Android runtime. It is to be noted that sending(receiving) of certain Intents (e.g. <`action=ACTION_SHUTDOWN`> or Intents with `"content:"` URIs in `Data` field) in Android are protected by permissions and when JJB sends these Intents `SecurityException`s are generated. JJB is able to handle these exceptions gracefully and we discard these from our results. At present we focus on crash failures as opposed to thread hangs due to their visibility and negative user experience.

We discuss our results from three perspectives: (i) prevalence of crashes caused in the application components due to the fuzzed Intents for the various types of components and different fault injection campaigns; (ii) distribution of uncaught exceptions thrown by components in response to the fuzzed Intents; and (iii) error propagation from a user-level application to the Android framework.

In general, Android 2.2 displayed many more crashes than Android 4.0 and components in all the versions were

vulnerable to NullPointerExceptions. It was possible to crash some components by sending them an implicit Intent that matched exactly with their Intent-filter (i.e. nothing other than the mandatory fields were specified). In Android 2.2, three of the application crashes caused cascading failures which eventually restarted the Android runtime. The Android Emulator also showed signs of stress-related failures, whereby, the system_server (the framework component that coordinates interaction between Kernel space and user space) restarted periodically after testing a fixed number of components. The system_server is a key part of the Android environment—it runs a host of essential services (Power Manager, Device Policy, Search Service, Audio Service, Dock Observer, etc.). A crash of the system_server kills all user level application and services and restarts the Android runtime.

Below we present our experimental results organized into three discussions.

A. Results for Explicit Intents

In Section III-B2, we described how we generated explicit Intents for four different fault injection campaigns. In FIC A we sent an invalid <Action, Data> pair to components, in FIC B we sent an Intent with either Action or Data blank, in FIC C random bytes were assigned to either Action or Data, and finally in FIC D random bytes were assigned to Extras values. During our experiments we found a large number of crashes—2148 in Android 2.2, 641 in Android 4.0, and 152 for Marketplace apps. One may argue that a comparison between Android 2.2 on a real phone and Android 4.0 on an emulator compromises the validity of our results. To verify this, we conducted a smaller-scale test of Android 2.2 on emulator and Droid and did not find any major difference. Our choice of Android 4.0 on emulator was driven by the lack of a physical device in a timely fashion. Even if results obtained from a physical device change from its emulator (i.e. absolute numbers of crashes change), it does not invalidate the general trends described in our results. Below, we present an analysis of the observed crashes.

1) Distribution of Failed Components: We define a failed component to be a program that crashes at least once during a fuzz injection campaign. Due to the nature of our Intent generation it is possible that a component fails repeatedly in one experiment where that component is targeted, e.g. an activity that dereferences Data field without null check will crash for all Intents that has a blank Data field. Counting such repeated crashes masks the actual number of faults at source code, therefore, for a fault injection campaign like ours, a better metric of a framework's reliability can be obtained by finding how many failed components it has. Table I presents the number of failed components for various types (Activity, Broadcast Receiver, and Services) in each of our experiments. The number at the top, under the component type represents the total number of components of that type, e.g., Android 2.2 has 297 Activities. The number in the column "#crash" denotes the number of components that crashed.

It is encouraging to see that in all cases but two, the percentage of failed components is less than 10. The percentage of failed components in Android 4.0 is generally lower than in Android 2.2, with the exception of Services. Across experiments, Activities display higher fraction of failed components in FIC A than the rest. However, this may also be due to the fact that FIC A sends nearly twice as many Intents than FICs B, C, and D combined. The high count of failed components across component types in FIC B is another key finding of our experiments. This indicates that many Android components do not perform null checks before dereferencing a field from an Intent and, therefore, are vulnerable to blank fields. This fact is also verified by our data in the next section.

The failure percentages of Marketplace apps are nearly identical to that of Android 4.0 components with the exception of FIC A for Activities and for Services, where Marketplace apps are significantly more robust. However, it was observed that 3 of the apps had at least one component that failed one or more experiments. Though our sample size for Marketplace apps (5) is too small to make any claims about general robustness of third-party apps, we expected the Top 5 to be more robust as they come from reputed vendors. This intuition is only partially borne out by the analysis results.

2) Distribution of Exception Types: To understand how well the Android framework handles exceptional conditions, we measured the distribution of exception types from failure logs. Here, we are focused on uncaught exceptions, because they result in the crashes. Since we are interested in measuring what percentage of all the crashes are constituted by a given exception type, here we count each crash individually. Thus, if in one experiment, 100 fuzzed Intents are sent to a component and the component crashes 20 times, we will have 20 data points (unlike in Section IV-A1 where we would have counted the component as having crashed and it would have resulted in a single data point). It can be seen from Fig. 3 that NullPointerExceptions (NPE) make up the largest share of all the exceptions. Though the percentage of NPEs in Android 4.0 (36.50%) has improved since Android 2.2 (45.99%), this is still significant and concurs with our findings in Section IV-A1. The results are given in terms of percentage of all the exceptions, thus for a given Android version, all the exceptions' numbers should sum to 100%. Other exceptions like ClassNotFoundException and IllegalArgumentException are significantly lower in Android 4.0 than in its previous version. Though exception types are sensitive to input data, we are applying similar inputs to the two different versions of Android. Therefore,

	Droid (Android 2.2)						Emulator (Android 4.0)						Marketplace Apps on Droid (Android 2.2)					
	Activities 297		Services 42		Broadcast Receivers 59		Activities 332		Services 54		Broadcast Receivers 69		Activities 103		Services 11		Broadcast Receivers 10	
	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%
A. Semi-valid	30	10.1	1	2.4	2	3.4	29	8.7	3	5.6	2	2.9	4	3.9	0	0.0	0	0.0
B. Blank	21	7.1	1	2.4	6	10.2	8	2.4	3	5.6	6	8.7	2	1.9	1	9.1	0	0.0
C. Random	18	6.1	1	2.4	4	6.8	9	2.7	3	5.6	2	2.9	2	1.9	0	0.0	0	0.0
D. With Extra	13	4.4	1	2.4	1	1.7	7	2.1	3	5.6	0	0.0	3	2.9	0	0.0	0	0.0

Table I
SUMMARY OF COMPONENT CRASHES IN DIFFERENT VERSIONS OF ANDROID IN RESPONSE TO FUZZED INTENTS IN FOUR DIFFERENT INJECTION CAMPAIGNS. HERE ONE COMPONENT CRASHING ONE OR MORE TIMES IN RESPONSE TO ONE OR MORE MALFORMED INTENTS DIRECTED AT IT COUNTS AS ONE CRASH.

our comparisons across the two versions are still valid.

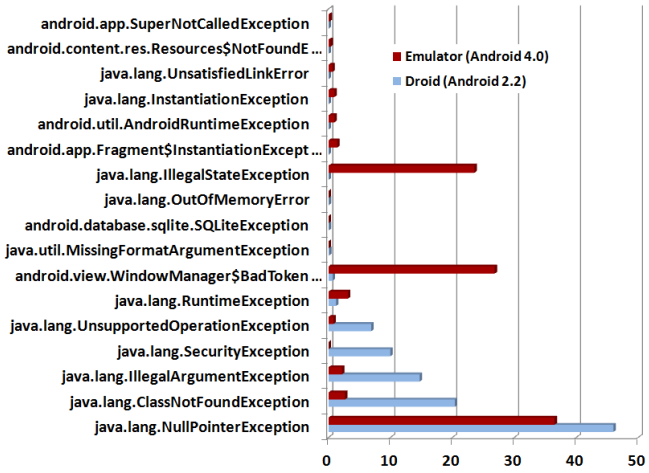

Figure 3. Distribution of different types of (uncaught) exceptions in Android 2.2 and 4.0. The bars represent percentage of all the exceptions, thus will sum to 100 (for each Android version). Note that we do not include Marketplace apps for this study.

However, the most significant finding from this study is the *introduction of unpredictable environment-dependent errors* in Android 4.0. Fig. 3 shows that the second, third and fourth largest exception types in Android 4.0 are `android.view.WindowManager$BadTokenException` (26.83%), `java.lang.IllegalStateException` (23.56%), and `java.lang.RuntimeException` (3.12%). These exceptions are almost non-existent in Android 2.2. A dominant reason for these crashes was garbage collection, where resources allocated to activities were released—a severe side-effect being restart of the Android `system_server`. It was observed that the same fuzzed Intent sent to the same component at a different time point in the experiment did not always cause the failure, or caused a different failure. The exact manifestation depended on the state of the device (the Emulator of the device to be more precise).

Another important point to note for Android 2.2 is the presence of exceptions that are typically thrown by the framework to notify the calling component of erroneous input or state, e.g., `java.lang.IllegalArgumentException`, `java.lang.SecurityException`, `java.lang.UnsupportedOperationException` etc. It is the responsibility of the calling function to implement proper exception handling, however such behavior is often missing in standard Android components.

3) System Crash from User Level Applications: Another significant discovery from our experiments was the *cascading failure of the Android runtime system*. We found a total of three Activities in the built-in applications that caused Android's `system_server` to restart. Due to the sensitive nature of these bugs and their potential security impact on millions of Droid users, we shall not disclose the names of the applications or the Activities in this forum. Instead, we use the generic name ActivityX for purposes of explanation. All of the failures occurred due to `NullPointerExceptions`. Upon inspection of the configuration files of these activities, it was revealed that all these activities run under the "system" process of Android (i.e. `system_server`). When these activities tried to access some fields inside an Intent, they did not catch the `NullPointerException`, which crashed the current thread and eventually sent Signal 9 (SIGKILL) to Android `system_server`. A special concern is that to test these components JarJarBinks did not need any extra permission at install time. Thus, potentially, any user level application is capable of sending the malformed Intents to these vulnerable Activities, causing the entire device to crash. Such promiscuous use of privileged operations is a concern for millions of customers using Android 2.2/2.3 handsets.

```
.............................
57   final Bundle extras = getIntent().getExtras();
58   mAccount = extras.getParcelable(EXTRAS_ACCOUNT);
.............................
```

Figure 5. Code responsible for crash of ActivityX, which eventually causes the entire device to crash

Let us take a look at the stack trace for one of these crashes. This crash occurred when we sent an Intent {`act=ACTION_PACKAGE_DATA_CLEARED cmp=android/.ActivityX`} to the Activity. The stack trace for this crash (refer Fig. 4) showed an error at line 58 of the source file `ActivityX.java`. The relevant code snippet is shown in Fig. 5. This code tries to read the extra field `EXTRAS_ACCOUNT`. However, since our Intent did not specify an `Extras` field, it raises a `NullPointerException`. This uncaught NPE kills the

```
.....................
I/ActivityManager(   62): Starting activity: Intent { act=ACTION_PACKAGE_DATA_CLEARED cmp=a
ndroid/.accounts.[redacted] }
W/dalvikvm(   62): threadid=7: thread exiting with uncaught exception (group=0x4001d800)
E/AndroidRuntime(   62): *** FATAL EXCEPTION IN SYSTEM PROCESS: android.server.ServerThread
.....................
E/AndroidRuntime(   62): Caused by: java.lang.NullPointerException
E/AndroidRuntime(   62):        at android.accounts.[redacted].onCr
eate([redacted].java:58)
E/AndroidRuntime(   62):        ... 6 more
I/Process (   62): Sending signal. PID: 62 SIG: 9
I/Zygote  (   33): Exit zygote because system server (62) has terminated
```

Figure 4. Partial stack trace of crash of ActivityX, which eventually causes the entire device to crash

thread of this activity and eventually the process, which, in this case, is `system_server`. The problem can be avoided by verifying that the `extras` object in line 57 is not null before accessing it, or by handling the exception gracefully. The severity of this bug lies in its ability to crash Android `system_server`, in other words, to render the device unusable till the Android runtime is restarted.

B. Results for Implicit Intents

In experiment A, we sent implicit Intents that applications had opted in to receive but we left all unspecified fields blank, e.g., when a filter only restricts the `Action`, there is no `Category`, `Data`, or `Extras` field set. Overall, the HTC phone had 211 applications registered from which we could derive 1910 Intent-filters. For each Intent-filter, we sent out exactly one Intent matching the filter through `startActivity()`. Note that some of these Intent-filters are registered by Services, hence, sending a matching Intent through `startActivity()` simply results in an `ActivityNotFoundException`. Those Intents that were delivered to an application, crashed 5 of the recipients. 12 unexpected exceptions occurred during the experiment, which are exceptions other than `ActivityNotFound` or any flavor of security exception. Most frequent exception was once again the `NullPointerException` followed by `IOException` and `Resources$NotFoundException`. All three are the result of insufficient input validation either causing a missing value to get dereferenced (NPE) or, even worse, propagated as an argument to a IO or resource loading call. At the end of the experiment, the phone crashed with a system reboot in 50% of the cases due to cascading failures. Even though the number of failures is not large relative to the number of applications tested, it has to be pointed out that all Intents we sent are completely *valid* according to what a sender is able to find out through the Intent-filters. *The problem arises from the fact that there is a significant amount of unspecified assumptions about the Intents that the receivers take for granted and fail to verify (e.g., a specific information in the `Extras` data being present).*

Experiment B goes a step further by combining all valid combination of `Action` and `Category`, thereby, significantly enlarging the number of Intents sent.

Table II
FREQUENCY DISTRIBUTION OF CRASHES WITH IMPLICIT INTENTS BY EXCEPTION TYPE

Exception Type	#Crashes
NullPointerException	32
IOException	22
RuntimeException	13
ArrayIndexOutOfBoundsException	6
android.content.res.Resources$NotFoundException	4
ClassCastException	3
TimeoutException	1
com.sprint.internal.SystemPropertiesException	1
IllegalArgumentException	1

From the Intent-filters, we were able to derive 643 distinct `Actions` and 37 `Categories` that were used in at least one of the filters. For each application, we now generated all possible combinations of `Action` and `Category` that were valid according to the filter. The experiment consistently crashed the phone after 26 out of the 211 applications tested. This happened even though we set the delay between the Intents to 2 seconds to allow for manual interaction (e.g., closing dialog boxes) and thereby avoiding resource exhaustion.

From this small set of 26 tested applications, we observed 83 exceptions. The distribution of the specific exception types is shown in Table II with `NullPointerException` and `IOException` again being the most frequent ones. Overall, 14 applications crashed during the experiment and showed a dialog to the user and only half of them were actually targeted directly, i.e., were the applications from which the filter was derived. The majority of the applications (including basic apps like Clock, Internet, Gallery, etc.) were most likely affected due to collateral failures, e.g., an Intent matching more than one filter and getting routed to more than one component.

C. Discussions

Our experiments have so far revealed three important aspects of Android—first is the presence of many components with poor exception handling code (most of these relate to `NullPointerExceptions`), second is the prevalence of environment-dependent errors in Android 4.0, and third is the presence of privileged components with unrestricted

access. The first problem can be addressed by a methodical training of developers on good exception handling practices. Application developers should always check for exceptional conditions when dealing with inputs (Intents) from external sources. Resolution of the second and third problems need more work at the Android framework level. The third issue also exposes some potential problems with Android's default policy for process-assignment of an application component. At present a component X in application A can run in the process of application B if A and B are signed with the same developer key. Despite signature-based permissions, this may pose a problem for vendors that build custom ROMs. If a component (C) of this custom build is permitted to run as privileged process, it may wreak havoc like ActivityX in a similar fashion (note that component C and the kernel of this build are signed with the same key). A potential solution is to restrict accessibility of component C with an explicit permission, in other words, every component running in a privileged process must be protected by explicit permissions.

JJB Limitations: Apart from its handling of new tasks and alert dialogues (where a tester must manually close these), JJB has another limitation—it cannot distinguish between thread hang, resource exhaustion, and UI wait. Detecting thread hangs in response to a malformed Intent would require knowledge of a component's life cycle which is currently not visible in logs generated by `logcat`. Our future work would look into adding this capability in JJB.

V. Android IPC Design Recommendations

The key challenge in making Intents more robust is the lack of a formal schema. Intents are effectively untyped; their application-level type is only determined by a String identifier but is not reflected by the Java type system. Therefore, there is no explicit contract between a sender and a receiver of an Intent and mutual agreement is expected among the two about what format of data a specific Intent needs to have and what an invalid message is. Additional data is stored in a map-like data structure that is not fully type safe either. The data structure keeps separate key spaces for values of different types and provides typed methods for adding and retrieving data but it is again not formally specified what the expected additional values are and which type they are supposed to have. It is up to the author of the receiver code to perform the input validation, which is a repetitive and error-prone task. To make matters worse, primitive types are stored and retrieved as actual primitives, which means that in the absence of the value the result is the neutral element of the type, e.g., `false` in the case of a boolean value. The absence of a primitive value in the extra data is therefore not detectable by the receiver. Another problem arises from software evolution. Implicit message formats are hard to keep consistent across different versions of the applications, especially within an ecosystem where components are contributed by different sources. There is no way to version a specific Intent or to indicate compatibility between a sender and a receiver.

A. Subtyping/POJO Approach

One way to make the message format more explicit and therefore possible to capture for an automated message verification system is to use subclasses for Intents instead of a single flat type. Extra data belonging to a message would be expressed as fields of the subtype. In the spirit of Plain Old Java Objects (POJOs), there would be getters and setters for the field. As a side effect, the Java compiler can now do automatic type checking since the messages use a type schema that the compiler is able to understand and enforce. What this approach does not achieve is further constraints on the values of data. For instance, there is no way to enforce a certain reference-type value to be not null or a numeric value to be always smaller than 10. Furthermore, there is currently no way in Java to express version information of classes in a standardized and accessible way. The cost for using the subtyping approach is that the total footprint of the platform is slightly increased since every Intent type now becomes a separate class in a separate file.

With a little experiment we found that a single class (subclass of Intent) with 3 fields (String, int, URL) having bean-like setters and getters adds 273 bytes to the footprint of an Android application, while the increase in size for a class with 6 fields is 403 bytes. Considering a handset where we have 200 Intent types, this implies a 80KB additional footprint for turning all these Intents into Subtypes with 6 fields. We argue that this is, in fact, an upper bound on footprint increase since we consider average 4-6 fields per Intent. In reality, most Intents have only between 2-3 fields, with few having a large number of fields (e.g., informative Intents like Battery Status).

B. Java Annotations

One way to express additional constraints about the message format when choosing the subtyping approach is the use of Java Annotations. Annotations are fully embedded into the language (since Java 1.5) and can be processed by the Java compiler. Therefore, it is possible to use the annotations already at compile time for criteria that are amenable to static checking. For dynamic checks, the corresponding code can either be realized as a common generic checker facility implemented as part of the Intent delivery mechanism of the platform or synthesized and injected into Intent receivers.

C. IDL and Domain Specific Language

Extended input validation requires additional knowledge about the message format since the semantic gap between the implicit message format and what can explicitly be expressed by classes and the Java type system is still large. For instance, an Intent responsible for a contact lookup might want to be able to do approximate matching and return

the contact names together with a matching factor between zero and one. In the Java type system, it would have to use a float type for the latter data but thereby would extend the range of permitted values to the entire IEEE 754 floating point number range. Another example is the problem that every reference type can always be set to null so that there is no way to express mandatory data in messages. One way to more expressiveness is to use a domain specific language to express the schema of the Intents.

Historically, a similar approach has been taken with many RPC systems which used an interface definition language (IDL). This IDL describes exactly the format of a remote invocation in enough detail so that the stub and skeleton code can be synthesized from this description. Systems like CORBA extensively used IDLs but arguably also web services employ the same principle, e.g., through the WSDL files. For instance, a type system like XML Schema allows value restrictions and would be a viable candidate for a domain specific language approach to specifying Intents. A well-designed domain specific language can express any type of constraint and therefore permit full input validation including version checks.

There are two different possibilities to interface general-purpose languages with domain-specific languages. External DSLs are free-standing and independent of the host language. IDLs, for instance, are external DSLs. As a result, however, code written in the host language and the meta-data written in the DSL have to be developed independently and cannot easily be cross-validated by existing tools. Internal or embedded DSLs are themselves implemented in the host language and therefore agree much better with existing tools. They are, however, restricted to what the host language can express.

VI. Related Work

Robustness evaluation of software systems is broadly categorized into functional and exceptional testing. Functional testing [17] employs generation of expected test inputs with the intention of checking the functionality of a software module, while exceptional testing employs generation of specially crafted test inputs to crash the system in order to check its robustness. Generated input test data can be random, a pure fuzz approach [6], or semi-valid (intelligent fuzzing) [10], [18]. UNIX utilities were first fuzzed by Miller *et al.* [6] by feeding random inputs to show that 25-33% of utility programs either crashed or hanged on different versions of UNIX. This simple technique has caught a variety of bugs like buffer overflows, unhandled exceptions, read access violations, thread hangs, memory leaks, etc. A later work by the authors [7] showed that robustness of UNIX utilities improved little over five years. A study [8] of similar nature on Windows NT and Windows 2000 showed their weakness against random Win32 messages, while, blackbox random testing on MacOS [9] reported a considerable lower failure rate (7%). Our research extends these works to a mobile platform where we fuzz the ICC of Android and show a variety of exception handling errors. In terms of knowledge about the target application (i.e. whitebox [18], [19] vs. blackbox testing [20]), our tool takes a combined approach (blackbox for explicit Intents and whitebox for implicit Intents).

Fuzz tools reported in literature can also be classified based on their input generation techniques and their intrusiveness. The input data produced by a fuzzer tool may be either generation based or mutation based [21]. Generation based fuzzers generate test inputs based on specification of a protocol or an API to be tested while mutation based fuzzers rely on capturing and replaying a mutated version of valid input. Our tool (JJB) falls under generation-based fuzz tools, as it generates input data, i.e., Intents conforming to Android Intent API specifications. JJB is also intelligent in that it has knowledge of Android APIs (e.g. known `Action`, `Category`, and `Extras` strings) and partial knowledge of the target applications (e.g. Intent-filters). Fuzzing tools typically produce input received across trust boundaries [22], i.e., Runtime-OS and Application-Runtime boundary. At a lower layer, fuzzing can be done at Runtime-OS interface as shown by [23]. Another similar work, Ballista [10], identified ways to crash operating systems with a single function call at Runtime-OS boundary. At a higher layer, fuzzing can be done at Application-Runtime boundary where runtime is responsible for validating data. In this work, we fuzz at Application-Runtime boundary with the aim of crashing Android runtime by fuzzing *Intents* that are passed between application components.

Fuzz testing has been employed in other domains like web applications, web servers, web browsers [24], Java-based applications [25] and SMS systems [26]. Fu *et al.* [25] presented an approach for compiler-assisted fault generation for testing error recovery codes in Java server applications. This is complementary to our work—the applications that had exception handling codes may be further evaluated by this tool, while in JarJarBinks, we found uncaught exceptions. Furthermore, JarJarBinks can additionally test Android market apps, for which source codes may not be available. We do not know any rigorous study of fuzz testing on smartphones. The closest work is [26], that fuzzes the messages going through the mobile telephony stack. They provide a fuzz based injection framework, that uncovers vulnerabilities on SMS implementation in smartphones, and can be abused for DoS attacks. In particular, the authors were able to crash iPhone applications and disconnect Android devices from mobile phone network. In our work, we evaluate a wider range of applications in Android and focus on Inter-component Communication.

A malformed Intent delivered to a receiver through ICC exposes attack surfaces as pointed out by [13], example vulnerabilities being triggering of components that are un-

intentionally exported by a developer (i.e., an Intent spoof) or unauthorized receipt of an implicit Intent by malicious component. ComDroid [13], a *static* analysis tool, detects these two vulnerabilities in Android applications. We narrow down these attack surfaces to a set of input validation errors by *runtime* testing, however, actual exploit of these errors may require combining these with other vulnerabilities (e.g. improper permission assignment). Our approach discovers vulnerabilities in the application components, but, we do not provide exploits to use these vulnerabilities from an external source, i.e., we do not show external requests that will generate malformed Intents for actually exploiting these vulnerabilities. That is part of our ongoing work.

Other work on Android security looked at permission assignment of applications, misuse of sensitive information [27], and provided future directions for application certification [28]. Our work does not directly detect privacy leaks, but can be used for giving insight to good application design practices (specially input validation). These practices in turn can be incorporated in an application certification process that is geared towards improving application robustness.

VII. Conclusion and Future Work

In this paper, we have successfully conducted an extensive robustness testing on Android's Inter-component Communication (ICC) mechanism by sending a large number of semi-valid and random Intents to various components across 3 versions of Android. Our learnings from this fault injection campaign are many, most prominent ones being: 1) Many components in Android have faulty exception handling code and `NullPointerExceptions` are most commonly neglected, 2) It is possible to crash Android runtime by sending Intents from a user-level process in Android 2.2, 3) Across various versions of Android, 4.0 is the most robust so far in terms of exception handling; it, however, displays many environment dependent failures.

Based on our observations, we have highlighted the guideline that any component that runs as a thread in a privileged process should be guarded by explicit permission(s). We have also proposed several enhancements to harden implementation of Intents; of these, subtyping in combination with Java annotations can be easily enforced. Our experiments have so far looked at robustness of Android components. In future we wish to explore whether any of the detected failures can be exploited by attackers, more specifically whether these failures can be triggered by an adversary who does not have physical access to the phone. Robustness evaluation of Binder IPC in Android is another future goal.

References

[1] D. Gross, "Fallout continues over smartphone tracking app," December 2011. [Online]. Available: http://www.edition.cnn.com/2011/12/02/tech/mobile/carrier-iq-reactions/

[2] K. LaCapria, "iphone explodes in midair on aussie flight," November 2011. [Online]. Available: http://www.inquisitr.com/163661/iphone-explodes-in-midair-on-aussie-flight/

[3] T. Lee, "At&ts samsung galaxy s2 security flaw lets you bypass the lock screen," October 2011. [Online]. Available: http://www.uberphones.com/2011/10/atts-samsung-galaxy-s2-security-flaw-lets-you-bypass-the-lock-screen/

[4] D. ben Aaron, "Google android passes 50% of smartphone sales, gartner says," November 2011. [Online]. Available: http://www.businessweek.com/news/2011-11-17/google-android-passes-50-of-smartphone-sales-gartner-says.html

[5] R. Golijan, "Fridge magnet poses security threat to ipad 2," April 2012. [Online]. Available: http://www.technolog.msnbc.msn.com/technology/technolog/fridge-magnet-poses-security-threat-ipad-2-119905

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32 – 44, December 1990.

[7] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," University of Wisconsin-Madison, Tech. Rep., 1995.

[8] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*. Berkeley, CA, USA: USENIX Association, 2000.

[9] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 78 – 86, January 2007.

[10] P. Koopman and J. DeVale, "The exception handling effectiveness of posix operating systems," *Software Engineering, IEEE Transactions on*, vol. 26, no. 9, pp. 837 – 848, sep 2000.

[11] "Dalvik virtual machine," 2008. [Online]. Available: http://www.dalvikvm.com/

[12] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, pp. 153 – 163, January 2008.

[13] 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*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239 – 252.

[14] "What is android?" [Online]. Available: http://developer.android.com/guide/basics/what-is-android.html

[15] "Intent fuzzer." [Online]. Available: http://www.isecpartners.com/mobile-security-tools/intent-fuzzer.html

[16] "Intent class overview." [Online]. Available: http://developer.android.com/reference/android/content/Intent-.html

[17] B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Verlag John Wiley & Sons, Inc, 1995.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[19] J. DeMott, "The evolving art of fuzzing," June 2006. [Online]. Available: http://www.vdalabs.com/tools/

[20] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ser. RT '07. New York, NY, USA: ACM, 2007.

[21] P. Oehlert, "Violating assumptions with fuzzing," *Security Privacy, IEEE*, vol. 3, no. 2, pp. 58 – 62, march-april 2005.

[22] J. Neystadt, "Automated penetration testing with white-box fuzzing," February 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc162782.aspx

[23] A. Johansson, N. Suri, and B. Murphy, "On the selection of error model(s) for os robustness evaluation," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, june 2007, pp. 502 –511.

[24] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional., 2007.

[25] C. Fu, A. Milanova, B. Ryder, and D. Wonnacott, "Robustness testing of java server applications," *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 292 – 311, april 2005.

[26] C. Mulliner and C. Miller, "Injecting sms messages into smart phones for security analysis," in *Proceedings of the 3rd USENIX conference on Offensive technologies*, ser. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009.

[27] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011.

[28] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235 – 245.