Intent Fuzzer: Crafting Intents of Death

Raimondas Sasnauskas
University of Utah
Salt Lake City, UT, USA
rsas@cs.utah.edu

John Regehr
University of Utah
Salt Lake City, UT, USA
regehr@cs.utah.edu

ABSTRACT

We present a fuzzing framework for Intents: the core IPC mechanism for intra- and inter-app communication in Android. Since intents lie at a trust boundary between apps, their correctness is important and thorough testing is warranted. The key challenge is to balance the tension between generating intents that applications expect, permitting deep penetration into application logic, and generating intents that trigger interesting bugs that have not been previously uncovered. Our work strikes this balance using a novel combination of static analysis and random test-case generation. Our intent fuzzer crashed dozens of Google core and top Google Play apps, resulting in app restarts or even in a complete OS reboot.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Experimentation, Reliability, Security

Keywords

Android IPC, fuzz testing, random testing, static analysis

1. INTRODUCTION

The ever-growing popularity of Android and its market share for mobile smartphones constantly attract new developers and businesses that target the huge user base. At the same time, sensitive data stored on devices induces attackers to search for new ways to compromise the system and steal data. In particular, the presence of malware apps [9] indicates the existence of a global market for users' private data.

A major focus of Android security research is analyzing apps for malicious behavior. The ubiquitous risk of privacy leaks demands rigorous techniques to early detect both intended and unintended exposure of users' private data, such as location, contacts, and photos, to third parties. Additionally, the variety of sensors and new input sources (NFC, QR codes, Bluetooth LE, camera, etc.) on modern phones increase the number of trust boundaries between the unprivileged inputs and the OS causing new security risks [10].

WODA+PERTEA'14, July 22, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2934-7/14/07...$15.00
http://dx.doi.org/10.1145/2632168.2632169

In this paper, we focus on *Intents*: the message-based IPC in Android. In our initial analyses, we observed that many Android apps export their components to unprivileged apps in the system, i.e., any app can explicitly trigger the exported component's execution via an intent (see Table 1). Consequently, we argue that the processing of intent's structured data at the app's trust boundary represents a security risk that has to be carefully addressed.

Generating highly structured inputs that get high code coverage for arbitrary closed-source apps in Android is an open engineering challenge. Particularly, the complexity of the Android OS makes it difficult to directly apply the well-established techniques such as concolic/symbolic execution. Conversely, we propose a runtime-efficient combination of static analysis and random fuzzing to dynamically exercise both open- and closed-source Android apps. We argue that this testing combination represents a good trade-off between soundness and runtime efficiency, while being effective in finding distinct crash bugs in numerous Android apps.

The remainder of this paper is structured as follows. Section 2 provides background information on Android's components and IPC. Section 3 describes our intent fuzzer techniques. We present the preliminary implementation and results in Sections 4 and 5, discuss the related work in Section 6, and conclude in Section 7.

2. ANDROID'S COMPONENTS AND IPC

Every Android app is composed of a set of *components* that encapsulate the app's logic. There are four types of components: *Activities* (a single UI screen), *Services* (background operations without UI), *Content Providers* (data query and modification), and *Broadcast Receivers* (response to system-wide broadcasts). Each component can be started explicitly by the system, hence, there are multiple entry points to the app's execution.

Table 1: Exported components of top Google Play free apps

App	Activities	Services	Providers	Receivers
Facebook 8.0	12	4	4	14
Pandora 4.5	4	0	1	5
Instagram 5.1	3	0	0	0
FB Msg. 2.2	3	0	0	3
Snapchat 4.1	1	0	0	0
Netflix 2.1	2	0	0	1
Candy CS 1.29	1	0	0	3
Kik Msg. 7.1	3	0	1	2
eBay 2.5	6	1	1	4
WhatsApp 2.11	11	2	0	4

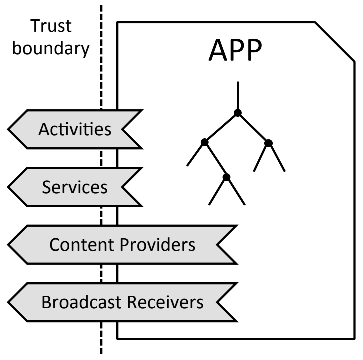

Figure 1: Android app's exported components

The declaration of the components and their capabilities resides in the app's manifest file `AndroidManifest.xml`. For example, next to the Java package name, the declaration may specify the component's requested *permissions* to access Android's protected API and permissions that other apps require to interact with the app's components. After the installation of the app, the declaration of the components cannot be changed during runtime.

The activation of components is triggered by intents: messaging objects that specify an action to be performed. Intents can be either *explicit*, targeting one specific component, or *implicit*, merely declaring a desired action (e.g., open a web page) without specifying which component should perform the work. Subsequently, the Android system finds the appropriate component for intent delivery. Intents embody the IPC in Android and are heavily used for both intra- and inter-app communication. Still, the primary use-cases of intents are starting an activity, service, or sending a message to the broadcast receivers.

There are two mechanisms in Android that control the access to the app's components from other apps. First, each component may specify its `exported` attribute in the manifest file. If set to *false* (default value), only the components within the same app are allowed to access the component. Otherwise, the component can be started from other apps via intents. Additionally, exported components are usually advertised using *intent filters* to specify the type of implicit intents they are capable to receive. Consequently, the presence of an intent filter in the manifest file automatically exposes the corresponding component to other apps in the system. Second, each component may specify the permissions the other apps must posses to allow the access to it.

Regardless the permissions, any exported components of an app denote the trust boundary for external data and actions (cf. Figure 1). As an example, the `ComposeActivityEmail` activity from Google's Email app is exported allowing access from any unprivileged system process in the system (cf. Figure 3). Additionally, the presence of an intent filter specifies which actions the component is able to perform on the expected data scheme.

3. FUZZER DESIGN

This section presents the design of the intent fuzzer and details on its core components. We describe the interplay between the static and dynamic analysis and discuss the features of the proposed technique.

The overview of the intent fuzzer is depicted in Figure 2. For each target app, the fuzzing workflow consists of five steps: (1) *component extraction* to identify the exported components and their actions, (2) *static analysis* to obtain the structure of the expected intents, (3) *intent generation* to create well-formed intents that trigger the actions, and (4) *data fuzzing* to randomly fuzz the intent data.

3.1 Component Extraction

As each of the app's components is described in the manifest file and cannot be altered after installation, it is straightforward to parse this information using e.g., Android SDK. Consequently, we collect the names and intent filters of all exported components. This data allows us to create explicit intents for the fuzzing phase that activate the components and match the advertised actions. Furthermore, the remaining fields in the intent filters may provide additional information about the structure of expected intents (e.g., data MIME type).

3.2 Static Analysis

The manifest information from the previous step describes the trust boundary between the components. However, this information doesn't provide any details about the structure of the intents that is processed during the execution of the advertised actions. For example, the `android.intent.action.SENDTO` action of the Email app's activity (cf. Figure 3) suggests that the action starts an UI to send an email, but there are no further details about the intent structure (e.g., e-mail subject, body, to-field, etc.). Although the Android SDK provides standard extra fields that can be used inside intents, we noticed that their usage is scarce in real-world apps.

In addition to the action and data expressed as an URI, any further information (extras) addressed to the component is stored in a *Bundle*. A bundle object is a mapping from string values to various data types that implement the *Parcelable* interface used in Android's IPC. For each such data type, Android's SDK provides methods to put (e.g., `putLong(String key, long value)`) and get (e.g., `getLong(String key)`) the extended data from the bundle object. The actual transmission of intents between the apps is implemented in the *Binder* kernel module that contains an efficient data copying using memory page flipping.

To automatically extract the extended information accessed by a component, our idea is to employ path-insensitive, inter-procedural CFG analysis. Starting from each component's entry point (e.g., `onCreate`-method for activities), we statically traverse the Dalvik bytecode collecting all calls to the intent objects' getter methods including calls to their bundle objects. The majority of the calls use a specific string key to extract the extra data from the intents whereas the data type itself is encoded in the name of the method. As each of the permitted method calls is documented in the SDK, we are able to uncover the complete superset structure of the intents that are processed inside the components. For example, our static analysis of the `ComposeActivityEmail` activity of the Email app returns the following results in JSON format where the different string keys denote several calls to the same getter method:

```
<activity
  android:name="com.android.email.activity.ComposeActivityEmail" ...>
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.SENDTO" />
    <data android:scheme="mailto" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
  </intent-filter>
  ...
</activity>
```

Figure 3: An exemplary activity from Google's Email app

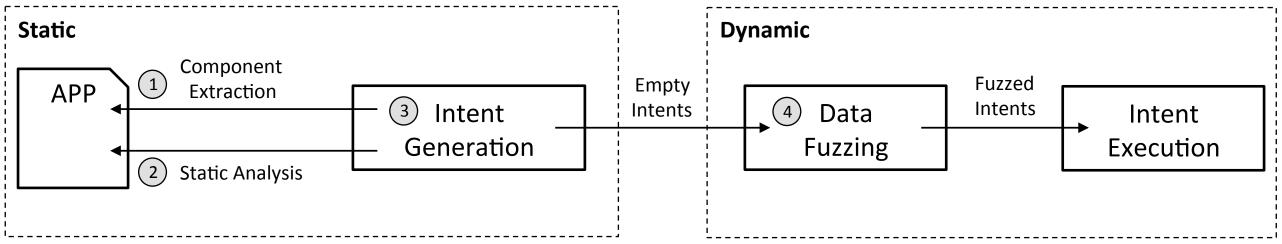

Figure 2: Intent fuzzer overview

```
"com.android.email.activity.ComposeActivityEmail":
{
  "getBundle": ["compose_state"],
  "getInt": ["action"],
  "getParcelable": ["account", "extraMessage",
    "in-reference-to-message", "extra-values",
    "android.intent.extra.STREAM"],
  "getParcelableArrayList": ["attachmentPreviews",
    "android.intent.extra.STREAM"],
  "getCharSequence": ["quotedText",
    "android.intent.extra.TEXT"],
  "getIntExtra": ["action"],
  "getParcelableExtra": ["original-draft-message",
    "in-reference-to-message",
    "in-reference-to-message-uri",
    "extra-notification-folder", "extra-values"],
  "getParcelableArrayListExtra":
    ["attachmentPreviews"],
  "getBooleanExtra": ["notification", "fromemail"],
  "getAction": [],
  "getData": [],
  "getExtras": [],
  "get": ["account"],
  "getStringExtra": ["account", "selectedAccount",
    "android.intent.extra.SUBJECT"],
  "getBoolean": ["showCc", "showBcc",
    "respondedInline"],
  "getStringArrayExtra":
    ["android.intent.extra.EMAIL",
    "android.intent.extra.CC",
    "android.intent.extra.BCC"],
  "getSerializable": ["attachments"],
  "getString": ["replyFromAccount",
    "fromAccountString"]
}
```

Compared to an intent without any extras that merely starts a component, the information returned from the static analysis is extremely valuable, especially for larger apps with lots of functionality.

In addition to the intent structure analysis, we collect all string constants during the component's CFG traversal. The idea of string collection is to use them as a seed from random string generation and mutation in the data fuzzing step.

3.3 Intent Generation

Given the information about the exported components and the structure of the intents they can be triggered with, the next step is to generate a set of empty intents a component is expecting to receive. For each intent action and data type (if any), we create an explicit intent object by specifying the component's name. Such a valid but "empty" intent ensures that the component gets activated during fuzzing reaching the code handling the specified action.

Afterwards, the dynamic analysis takes over entering a loop: For each valid intent, the information from the static analysis is used to fill the intent's structure with random data using the corresponding setter methods. Specifically, for each getter method, the corresponding setter method is called injecting some random data depending on the type. For example, for the `getInt("action")` call on a bundle object, the fuzzer calls `putInt("action", random_int)` with the same key and some random integer value updating intent's bundle object. The map-like superset structure of extras generated during the static analysis step turns out to be advantageous: even if the target component doesn't access one particular extra field during its execution, the unused entry doesn't influence the execution.

3.4 Data Fuzzing

To create random data for data types ranging from primitive types to objects, we employ the generative approach of QuickCheck [4]. This works well for primitives (e.g., int, long, char, String) and their composites (e.g., lists, arrays). For example, to generate a random value for the `getIntegerArrayList`-call, we implement a combined generator:

```
static ArrayList<Integer> fuzzIntegerArrayList() {
    return
      (ArrayList<Integer>)lists(integers()).next();
}
```

Similarly, for strings generation we seed the generator with the strings collected from the static analysis step. We thereby increase the chances to explore more branches of the code where the conditional is based on string comparison with the string constant from the intent.

However, many components process complete objects received via intents (e.g., `getParcelable`-call) that are tedious to generate manually. For each new object type, a new combined generator has to be implemented. Nevertheless, since the classes implementing the Parcelable interface have well-defined methods to pack and unpack their object data before transmission, the static analysis can be extended to automatically extract these method calls as well. As a result, an automated generation of random objects becomes feasible using QuickCheck's combined generators.

3.5 Intent Execution

Lastly, each new instance of an intent with fuzzed data is explicitly sent to the target component for execution. Upon delivery, the component is first restarted and does not depend on previous executions. During intent execution, we monitor both code coverage (open-source apps only) and crashes.

4. FUZZER IMPLEMENTATION

The prototype of our intent fuzzer is implemented as an extension to Android's GUI fuzzer Monkey [5]. The fuzzer can target either an emulator or a real device. For emulators, we experiment with both ARM and x86 targets on the most recent Android version (Android 4.4). For performance reasons, our primary target remains to be the KVM-enabled x86 emulator image. The performance gain using x86 compared to ARM emulation is remarkable allowing us to inject intents at a faster rate. However, many popular apps contain ARM native libraries and thus cannot boot on x86 or don't work properly.

For off-line static analysis of the app's APK file, we modified FlowDroid [1], a static taint analysis tool for Android that is based on the Soot [6] Java analysis framework. Instead of tracking taints, we instruct FlowDroid to track the information that reveals the structure of the intents processed by the exported components. To this end, we track the calls to all getter methods of the Intent and Bundle classes. The analysis dumps the results to a JSON file that is subsequently uploaded to the SD card on the running device. For each app, the static analysis is required only once.

During initialization, the fuzzer first parses the target app for exported components creating a set of valid intents. Second, it looks for static analysis information (if any) on the SD card and then starts generating fuzzed intents using the QuickCheck framework for Java [11]. We implemented the generation of primitive and composite Bundle types including URIs, but currently there is no support for Parcelable objects.

Finally, fuzzed intents are sent to the target app using Android's SDK and their processing is monitored for crashes and achieved coverage. To measure coverage, we first instrument the apps using Emma [7]. Second, we periodically dump the coverage data at runtime to the SD card. However, Emma requires the source code of the app and thus we cannot monitor the coverage of the closed-source apps where only the Dalvik bytecode is available. To parallelize testing, we created a scripted infrastructure that distributes the emulator instances among the available CPUs, runs the fuzzing campaign limited by number of fuzzed intents, and collects the error logs.

We treat error logs to be identical if the error message and the location of the failure are identical. To avoid false positives, we validate all crashes on a non-rooted physical device (e.g., Nexus 7) by injecting the corresponding intent using the `"adb shell am"` command line tool that runs as an unprivileged user on the device. Herewith, we validate that the crashes can be repeated from any unprivileged app in the system.

5. RESULTS

Table 2 provides details on the static analysis runtime per app. Except of WhatsApp, the analysis runtime stays below one minute. WhatsApp's components contain significantly more strings than other apps whose extraction results in a longer analysis time. Note that all apps are analyzed only once before fuzzing.

Initially, we enabled context-, object-, and flow-sensitivity in FlowDroid, but our analyses didn't scale for real-world apps. In particular, for intent processing analysis we require to hold all the explored paths in memory. To this end, FlowDroid runs out of memory for larger apps even on a 120GB machine. In addition, to soundly track an intent object and its subobjects we must use a sufficient access path length in FlowDroid. However, larger values (e.g., 3) make the analysis more expensive and didn't scale in our experiments. Disabling these features and using CFG parsing instead over-approximates the results (e.g., we see getter methods from intent objects that are not directly related to the incoming intents), but this additional information doesn't falsify the fuzzed intents.

We ran our intent fuzzer against all exported Android core apps and a number top Google Play apps (e.g., Facebook, Twitter, eBay, Amazon, etc.). For the majority of the apps, at least one of the exported components crashes with the `NullPointerException` when the data or other fields of the intent object are null. For example, the Email app can be crashed by starting the `FolderPickerActivity` component without any extras using the following command issued from the host:

```
$ adb shell am start -n
  com.android.email/.provider.FolderPickerActivity
```

In this example, the component accesses a field of a null-object, i.e., it does not check if the object retrieved from the intent is null. After a crash, the affected component is automatically restarted by the system. However, in case of an Android core service, such a crash leads to a reboot of the device representing a local DoS. For example, the following command triggers an uncaught `android.util.SuperNotCalledException` in the main Android service leading to a system reboot:

```
$ adb shell am start -n
  android/com.android.internal.app.ChooserActivity
```

These findings validate the reports of earlier studies [12] that the exception handling code of many apps remains to be poor. All tests were performed on a x86 Android 4.4 emulator image. We verified the failures on real devices and reported them to Android developers at Google.

In addition to the large number of null-input crashes, we observed that the injection of extra intent fields hits more paths in the code compared to intents that merely trigger the component's action. For example, the eBay app crashes with an `ArrayIndexOutOfBoundsException` when processing the following intent that contains a combination of three extra fields automatically generated by the fuzzer:

```
$ adb shell am start -a android.intent.action.SEARCH
  -n
  com.ebay.mobile/com.ebay.motors.QuickSearchHandler
  -e query not%20set --el
  com.ebay.mobile.Perspective.searchSandboxCategoryId
  -4611270473092182423 -e intent_extra_data_key
  user.sell
```

Leaving out one or two of the three extra fields results in an `NullPointerException`, but the array access violation is triggered only with a combination of the three fields. Thus, the fuzzer covers more code resulting in higher chances to uncover interesting corner-case bugs in deeper execution paths.

Since the intent processing code constitutes merely a small part of the whole app, the overall coverage is expected to be low. For example, we injected 100 fuzzed intents into the Email app and measured the coverage. Compared to the null intent fuzzer [2] (no data, no extra fields, valid intents only), our intent fuzzer increased the coverage from 8% to 9%. Still, this 1% increase constitutes an additional coverage of 15 classes, 166 methods, 3843 blocks, and 801 lines of code. In this work, the primary goal of coverage analysis in open-source apps is to get feedback about the code locations where the fuzzer performs poorly in terms of structured data generation.

6. RELATED WORK

In this section, we relate our fuzzing approach to research efforts that target the inter-component communication in Android. Therefore, we leave out any Android work that addresses other attack vectors and their countermeasures such as malware detection, kernel fuzzing, permission analyses, etc.

In a study of Android application security [8], Enck at al. identified the existence of unprotected broadcast receivers vulnerable to intent injection attacks and discussed potential exploitation techniques. ComDroid [3] by Chin et al. is one of the first Android static analysis tools that automatically detects application communication vulnerabilities. In their paper, the authors first categorized the intent-based attack surfaces such as unauthorized intent receipt (e.g., activity hijacking) and intent spoofing (e.g., malicious broadcast injection). Second, they implemented static analysis of apps augmented with limited inter-procedural analysis to track the properties of intent objects (whether an intent object has been made explicit, whether it has action, whether it has any flags set, and whether it has any extra data). Subsequently, ComDroid issues a warning if a potential vulnerability is detected. In contrast to ComDroid, our intent fuzzer generates concrete intent objects that lead to application crashes or potential vulnerabilities. Moreover, by leveraging the features of FlowDroid (e.g., by using the component life-cycle aware inter-procedural analysis), we penetrate deeper into the application revealing more information on the structure of the processed intents.

Our work was inspired by the null Intent fuzzer [2]—an unprivileged app that injects valid intents with the blank data field to other apps' exported components. Similarly, the authors of an empirical study of the robustness on inter-component communication [12] by Maji et al. extended this basic intent fuzzer in numerous ways and come closest to our work. For each target app, their fuzzer creates a set of valid and semi-valid intents with object fields selectively left blank. Additionally, they added standard extra data fields from the Android documentation to the intents expecting the apps to process them. The evaluation results show a large amount of crashes that mostly relate to `NullPointerExceptions` indicating poor exception handling code. Our experiments validate that this is till the case in Android 4.4 representing a source of potential local DoS attacks. One major advantage of our approach is that instead of adding random extras to the intents we are able to construct the expected intent structure allowing us to explore more execution paths. Also, unlike using purely random data, the generative approach of QuickCheck allows us to automatically create composite objects and seed the random generation process with data values collected statically (e.g., strings).

For URI data fuzzing, the authors of DroidFuzzer [13] focus on activities that process MIME data (e.g., "video/*") passed via an URI. For each MIME data type an activity expects, the tool continuously injects intents with abnormal data generated from a normal data seed using mutation. The processing of such data can lead to app crashes at both Java and native library level. Fuzzing URI-linked data with explicit MIME types is orthogonal to our approach. Conversely, we fuzz the URI string itself as any other field of the intent object or its extras.

Table 2: Static analysis runtime results

App	File size (MB)	Runtime (s)
Facebook 8.0	16.4	20.6
Pandora 4.5	6.4	14.2
Instagram 5.1	8.8	16.7
Facebook Messenger 2.2	5.6	10.1
Snapchat 4.1	7.7	15.3
Netflix 2.1	11.9	4.4
Candy Crush Saga 1.29	39.6	4.5
Kik Messenger 7.1	12.2	17.3
eBay 2.5	10.0	18.6
WhatsApp 2.11	14.5	428.9

7. FUTURE WORK

The key research challenge of this work remains to penetrate deep into application logic and uncover interesting bugs. To this end, we strive for runtime efficient methods accepting soundness and precision loss. We believe that adding automated fuzzing of Parcelable objects is one of the major steps to reach our research goal. These objects are frequently exchanged between apps necessitating thorough testing of the related object handling code.

Some apps process specific data (e.g., images, audio) from intents in their native libraries. Since crashes in native code are potentially exploitable, we plan to complement our work with data fuzzing techniques that go beyond the generation of fuzzed Java objects and primitive data types.

8. ACKNOWLEDGMENTS

We thank Scotty Bauer for his comments and technical assistance with the Android experiments. This research is being supported by the NSF grant CCF-1218026.

9. REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexander Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *To appear at PLDI*, 2014.

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

[3] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proc. MobiSys*, pages 239–252, 2011.

[4] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ICFP*, pages 268–279, 2000.

[5] Android Developers. Monkey tool, 2014. `https://developer.android.com/tools/help/monkey.html`.

[6] Soot Developers. Soot: a Java optimization framework, 2014. `http://www.sable.mcgill.ca/soot/`.

[7] EMMA Developers. EMMA: a free Java code coverage tool, 2014. `http://emma.sourceforge.net/`.

[8] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proc. USENIX Security*, pages 21–21, 2011.

[9] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2011.

[10] MWR InfoSecurity. Android NFC vulnerability, 2012. `https://labs.mwrinfosecurity.com/blog/2012/09/19/mobile-pwn2own-at-eusecwest-2012/`.

[11] Thomas Jung. QuickCheck for Java, 2014. `https://bitbucket.org/blob79/quickcheck`.

[12] 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*, pages 1–12, 2012.

[13] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proc. MoMM*, pages 68–74, 2013.