Template-based Android Inter Process Communication Fuzzing

Anatoli Kalysch
Friedrich-Alexander University
Erlangen-Nürnberg (FAU), Germany
anatoli.kalysch@fau.de

Mark Deutel
Friedrich-Alexander University
Erlangen-Nürnberg (FAU), Germany
mark.deutel@fau.de

Tilo Müller
Friedrich-Alexander University
Erlangen-Nürnberg (FAU), Germany
tilo.mueller@cs.fau.de

ABSTRACT

Fuzzing is a test method in vulnerability assessments that calls the interfaces of a program in order to find bugs in its input processing. Automatically generated inputs, based on a set of templates and randomness, are sent to a program at a high rate, collecting crashes for later investigation. We apply fuzz testing to the inter process communication (IPC) on Android in order to find bugs in the mechanisms how Android apps communicate with each other. The sandboxing principle on Android usually ensures that apps can only communicate to other apps via programmatic interfaces. Unlike traditional operating systems, two Android apps running in the same user context are not able to access the data of each other (security) or quit the other app (safety).

Our IPC fuzzer for Android detects the structure of data sent within Intents between apps by disassembling and analyzing an app's bytecode. It relies on multiple mutation engines for input generation and supports post-mortem analysis for a detailed insight into crashes. We tested 1488 popular apps from the Google Play-Store, enabling us to crash 450 apps with intents that could be sent from any unprivileged app on the same device, thus undermining the safety guarantees given by Android. We show that any installed app on a device could easily crash a series of other apps, effectively rendering them useless. Even worse, we discovered flaws in popular frameworks like Unity, the Google Services API, and the Adjust SDK. Comparing our implementation to previous research shows improvements in the depth and diversity of our detected crashes.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Penetration testing**;

KEYWORDS

Fuzzing, Android Security, Inter-Process Communication

ACM Reference Format:
Anatoli Kalysch, Mark Deutel, and Tilo Müller. 2020. Template-based Android Inter Process Communication Fuzzing. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3407023.3407052

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-8833-7/20/08…$15.00
https://doi.org/10.1145/3407023.3407052

1 INTRODUCTION

Fuzzing is an automated software testing technique based on the idea to provide randomly generated inputs to programs. Any program or library that processes input data can be fuzzed, e.g., exported APIs, reading files, user input fields, and network communication [14, 15]. Fuzzing has established itself as a viable addition to manual testing due to its high automation ratio – once a fuzzer is up and running, it can be left constantly looking for bugs with no manual interaction needed.

In this paper, we propose an automated template-based fuzzing approach for Android's IPC mechanism. IPC messages on Android are called *Intents*. We implemented an Intent fuzzer that targets all publicly exposed interfaces of an app, and integrated it into the open-source security audit tool drozer [11]. Contrary to previous work about fuzzing of Android IPC messages [3, 19, 22], we rely on an extensive pre-analysis of the app components structure to factor in the architecture of the app and fuzz with a better understanding of the conditions needed to improve the code coverage of our template-based Android IPC fuzzer.

Fuzzing 1488 apps, we found 450 apps that crashed completely during testing, including famous apps and widespread libraries such the Google Services Framework. 921 apps had at least one component that raised an exception during fuzzing, and we found 635 different components that can completely crash an app. This is particularly serious on modern OS like Android that enforce strict sandboxing between apps running in the same user context. Normally an Android app is not able to affect the state of another app in any way, neither by accessing it's data, nor by quitting or crashing it. Thus we show that numerous of crashes subvert some of the safety guarantees of the Android ecosystem, thereby endangering apps.

Note that we focus on the safety aspect of an app, especially on the code stability of the interfaces an app exposes to other apps. We determine common exception types and design flaws causing an app to crash, but no security implications such as shellcode execution. We did not investigate whether a crash can be exploited for two reasons. First, unlike C, Java is secure against code execution attacks such as stack overflows, leading to a very small number of crashes that can be exploited in Java. Second, we focus on automated fuzz testing, and additionally look into crashes manually to find common reasons for a crash, but judging whether a crash can be exploited or not requires reverse engineering and gaining a deep understanding of an app's logic.

We reported many of our findings to the developers. For example, we reported several bugs in the Google Play Services in early 2019 and most of them have been fixed starting January 2020.

2 BACKGROUND

Android's IPC model differs from common desktop operating systems. Direct process to process communication was traded in favour of a message passing approach, realized by Android's binder, which acts as an intermediary between apps.

Binder. The core of Android's binder is its kernel driver, it handles all communication between different processes. Inside the processes all communication is done by the binder API, which is part of Android's system APIs [16]. A message sent from a client process to a service is called a transaction. Each transaction which can be resolved by the kernel driver results in a method invocation by the service. Additionally, each transaction can contain a payload, stored in a `Parcel` container [21]. A transaction can be executed unidirectional or bidirectional. Bidirectional transactions expect a response from the service and are therefore executed synchronous. Unidirectional transactions are executed asynchronous [21].

Intents. For the apps themselves the previously described binder transactions are abstracted through *Intents*. Every Intent declares a recipient and contains data optionally. Therefore, every Intent can be seen as a self contained object which invokes parts of other apps and provides a set of parameters which can be used by the receiver [8]. The Intent object offers some standard fields for data, but most of the time the payload will be sent using a `Bundle` object. A `Bundle` in the Android system is an object containing a set of key-value-pairs. These mappings make it very easy for the part of the app the Intent is targeting to extract specific data [8]. A distinction is made between the explicit and implicit use of Intents. An explicit Intent specifies that it shall be delivered to a particular app defined in the Intent object, whereas an implicit Intent only specifies a certain type of operation which shall be fulfilled that will be executed by any app supporting this action [8].

Components. We focus the scope on the main components that usually create or process Intents: Activities, Services and Broadcast Receivers. An Activity describes one specific thing a user can do in an app. It interacts with the user and therefore almost always provides a graphical user interface and are managed through a UI stack. A new Activity can be created or called upon through Intents. The restriction for Activities is that they are regarded as foreground processes, meaning there should only be one Activity running at a time on the Android system. Background operations should be executed through Services. A Service does not provide a user interface, yet it can be started by another component and it will keep running even if the user switches to another app. Event-based operations are implemented through broadcast receivers which can receive broadcast messages from the operating system or other apps. These broadcasts are used to propagate information through the system and trigger actions [17].

Exports. After declaring components as exported they can be invoked from outside the scope of their app either by a direct or indirect Intent [10]. By using Intent filters the app's programmer can specify what type of indirect Intents shall be delivered to a component. Android's binder then automatically only delivers indirect Intents to the components matching the constraints specified in the filter. To protect components against resource access from malicious actors, Android offers the components to restrict access to them through custom permissions. This can be achieved either by setting permissions in the manifest file, specifically for one component, or as a default requirement for all exported components in the whole app.

3 DESIGN AND IMPLEMENTATION

This section details the design considerations and implementation of our approach. Note that we have open-sourced our implementation under the MIT license [4–6]. Figure 1 shows the architecture of our fuzzing approach. An APK file is parsed initially by the static analyzer module, which analyzes the exported Intent filters and creates a templates for valid intents for each found exported component. Based on these templates, the mutation engine starts the mutation pass and creates additional Intents that are stored in the Intent database. This leads the drozer module to install the APK on the dedicated device and initiate the fuzzing run, while the device logs are monitored by the log parsing engine and the crashes, exceptions, and their respective stack traces are saved to the crash database.

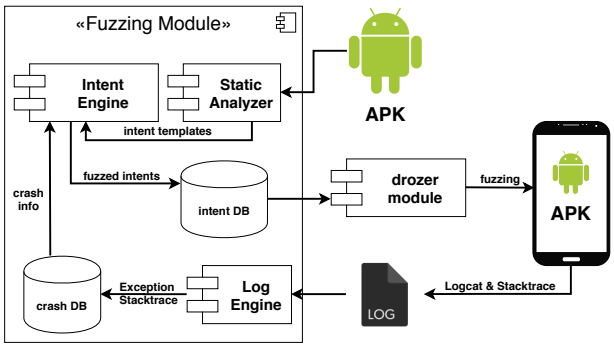

Figure 1: The architecture of our fuzzer. The modules for the static app analysis, intent engine and log parsing constitute the core architectural components, and instrument the drozer module to fuzz apps on the Android device.

As long as an Intent's payload can be properly flattened and unflattened by the Binder API and the Intent is well defined it is sent. This means a sender can potentially put any form of data in an Intent as long as he provides proper flattening for it. We take advantage of that. The idea is to build on empty Intents, extract from the application which data fields are used in further processing and which are not, and fill it with random data or data which is likely to be malicious. Examples for such data are null references, random data, range exceeding data or data which does not match the data type expected by the receiver. To get the best possible results, it makes sense to structure the payload of the Intents in a way that they match the structure the targeted components expect to receive.

Static Analysis and Template Creation. We start our static analysis at the manifest level, to detect all exported components and their Intent filters. As Intent filters describe what kind of actions, data URI patterns and categories a component claims, it is very likely

that by sending Intents meeting these resolution rules the targeted components can handle them and execute into a deeper path. To test the behaviour of components against malformed incoming Intents as intensive as possible, it is necessary to trigger as many execution paths as possible during a fuzzing campaign. Therefore, an appropriate structure of the fuzzed Intents is very important.

Building upon the rules from the static analysis we build valid Intents and start mutating them. For certain fields the Android manifest gives hints on how parts of the Intent should look like, e.g., for data URIs. Others, like the extra field, store the payload of an Intent as key-value-mappings – without the knowledge of which Intents will not penetrate beyond the unpacking class and might not even throw an exception. Unfortunately, the manifest file contains no information regarding the structure of these mappings, so a source code analysis becomes necessary. Same is true for the category fields of Intents.

Fortunately, the `Intent` class, has defined set of methods used to extract mappings from an Intent's extra field, making it easy to track all invocations of Intent related methods and thereby get a list of all mappings expected by each component. Starting from the entry point of each exported component we thus search for these mappings, creating dependencies between Intents and the expected fields. From these dependencies we create Intent templates, of how a valid Intent for each exported component should look like, i.e., which fields are required, how key-value-mappings look like, and what data types they are unpacked to. With this information our Intent Engine starts the Intent building and mutation processes.

Intent Building. From the provided templates we first generate valid Intents, filled with the expected data types. The action, data, category, and extras fields of Intents appear to be the most relevant ones, as they are frequently use by most components and their absence often leads to handled exceptions. Then, we mutate all Intent fields that are processed or queried by the app. These mutations range from changes in the data itself to changes in the data types, null references, and randomized data.

Mutation Engine. The mutation engine our module uses is build in a modular way and easily replaceable with that from different fuzzers, e.g., AFL [23]. For this work we tested a simple implementation based on the `Random` class from the Android API for random primitive data is generation. The class provides a randomization engine using a linear congruential formula seeded with a 48-bit value. Additionally, we included `Radamsa` [9] to allow a more seasoned mutation approach to be used in conjunction with our tool. In a similar way other engines can be integrated.

After a set of templates was created they can be used to generate actual Intents by filling them with fuzzed data. After that the Intents can be sent to a targeted application. To achieve this we rely on drozers agent application installed on the test device which is used to send the Intents and interact with the Android OS.

We also prevent previous executions from affecting new fuzzing passes. For activities, Intent flags are used to make sure that every time an Intent is starting an activity, the activity is started in a new task. Services are stopped explicitly from another component using its application's context. Broadcast receivers are registered in the Android system when the application is installed. They act as separate entry points to the application and are therefore not dependent on the application's context. By setting flags for activities, stopping services before starting them again and adding a time interval between the execution of two Intents, we can assure that Intents generate results as independently from previous executions as possible.

Logfile Parsing. During all fuzzing runs Android's application and crash logs were collected. The crash logs list all exceptions related to application crashes including their complete Java stacktraces. We convert the stream of log entries, retrieved from a test device, into an ordered representation, and store its output in the crash database. The exceptions are grouped by the affected components, so that for each component a table exists with the stacktrace for every exception and the corresponding Intent that caused said exception. Furthermore, general information about the app, like its package name or the number of exported components is stored as well.

To help the parser navigate through the logs and provide additional information, we add new entries to the Android logs at the beginning and end of each fuzzing pass. The entries contain information about the current pass and the app, and each sent Intent. Every time it executes an Intent, the fuzzer logs its string representation. This is helpful for manual verification of crashes found during automated fuzzing.

4 EVALUATION

This section presents the evaluation of our framework with 1488 apps from Google Play. We used a Nexus 5X running Android 9.0 for all fuzzing tests. The dataset was divided into batches of 10 apps, then each batch was subsequently installed and fuzz tested. A test run for one app lasted 10 iterations, with each exported component being executed with all prepared Intents. During the run the app and crash logs from the test device were retrieved to monitor app crashes and exceptions. For the detected crashes we followed up with a manual log analysis and the reverse engineering of affected crashed components to review the reasons why the crashes happened.

Fuzzing Results. Out of the 1488 apps, 921 apps had at least one component that crashed with an exception during fuzzing, and 450 apps crashed completely during testing. We discount exceptions raised by the base class handling the Intent, and only regard exceptions raised in any other class than the first Intent receiving class. Crashes are counted independently of the point of origin, the only prerequisite being that the app process needs to crash with an uncaught exception. We treat crashes originating in the developer code and the crashes originating in third party frameworks differently, since the latter affect all apps using vulnerable versions of the framework. In this section we detail the exceptions and components found in the original developer code, while in section 4 we regard libraries and frameworks.

Exported Components. In general the results show no direct correlation between the number of exported components and detected crashes. Apps with a higher number of exported components did not necessarily show more crashes than apps with a lesser number. However, exposing larger parts of an app increases its attack surface

and therefore the number of possible paths accessible from outside the app. This leads to a higher risk for unsafe source code being executable from outside the app's scope. Overall we discovered 635 vulnerable exported components that could be used to crash 450 apps from our dataset simply by sending out an intent.

While *Services* constituted roughly 10.31% of all detected exported components they were not responsible for any of the detected crashes. *Broadcast Receivers* on the other hand while constituting 16.68% of exported components were responsible for 32.51% of detected crashes. Taking a deeper look into the crashed components we discovered that most were meant to interact with system broadcasts, having safeguards for these cases but no security mechanisms in case of absent or corrupted data being transmitted through an intent. *Activities* made up 73.01% of exported components and had very diverse reasons for app crashes. While previous studies [12, 19, 22] already discovered `NullpointerExceptions` to be the biggest issue with Activities, we found that many more additional cases and exceptions need to be taken into account when designing secure exported components. The following section 'Most common exceptions and crashes' has a detailed overview of the exceptions encountered in all exported components.

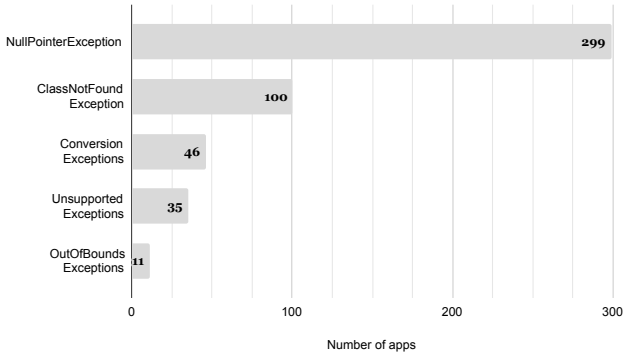

Figure 2: Most common exception types that generated an app crash. `NullpointerExceptions` **appear to be the biggest issue with components, followed by** `ClassNotFoundExceptions` **and several exceptions from wrong conversion.**

Most common exceptions and crashes. We list in table 2 all exceptions from our fuzzing results which caused an app to crash, ordered by the frequency of their appearance. Whenever key-value-mappings were left out or null references were assigned to the Intents, it was more likely that a `NullpointerExceptions` was raised by the receiving component. However, in case all expected mappings were assigned correctly but the format of the data stored in the mappings was modified instead, it was more likely that the payload passed the extraction step without any errors. This left a higher chance of discovering interesting bugs in deeper levels of the components' execution paths and resulted in more advanced exception types like `ClassNotFoundExceptions`. These arose when the URI scheme encoded in an Intent had some information about which class was needed to process the data in this Intent. Depending on the permissions of the app, an attacker can abuse this mechanism to infer sensitive information on the device, or even create agents for a reflective DDoS attack by proxy.All an attacker needs for these attack vectors is for his app to be installed on the victims device.

Further, we grouped following Exceptions by the categories 'Conversion Exceptions', 'Unsupported Exceptions', and 'OutOfBounds Exceptions' for figure 2. Conversion exceptions include the `ClassCastExceptions`, `IllegalArgumentExceptions`, and `NumberFormatExceptions`, all raised when conversion of data from of our fuzzed key-value-mapping did not succeed somewhere deeper in the app logic. Usually, they are raised in case a data conversion from one type to another has failed. For `ClassCastExceptions` this means that a component tried to convert an object to another type but the targeted type is not a subtype of the current object. `NumberFormatExceptions` are raised whenever a component has attempted to convert a string value to a numeric type, supported by Android, but the string does not have an appropriate format.

Unsupported exceptions include the `IllegalStateExceptions`, `InvocationTargetExceptions`, and `UnsupportedOperationExceptions`, which usually were the result of data types parsed the right way but interpreted wrong by the apps own logic. The tests showed that the main reason causing components to crash with these exception types, is that the content of input provided by Intents is very often not verified to have a correct format. Instead, input is in many cases handed down to the underlying routines of the apps directly by the components.

OutOfBounds exceptions was a category combining exceptions that relied on Intent data to perform array operations, e.g., `ArrayOutOfBoundsException`, `StringIndexOutOfBoundsException`, or the `IndexOutOfBoundsException`. Often raised by components that tried to read elements from an array or a collection based on an index, they are a sign of wrong assumptions about origins of incoming Intents.

Libraries and SDKs. During our experiments we noticed that not all crashes were caused by bugs in the developer code. Third party frameworks, functionality that is conveniently included into an app can have serious implications for the apps IPC. Three of the most noticeable ones are:

(1) *Google Play Services* offers a wide variety of different APIs which can be used by developers to integrate them in their apps. During our evaluation several different components from these APIs were detected creating crashes in apps exporting them. Interestingly enough, most issues arose with `BroadcastReceivers`. We found the `AnalyticsReceiver`, `GemReceiver`, the `CompaignTrackingReceiver`, and others to be vulnerable to `IllegalArgumentExceptions`, `IllegalStateExceptions`, and `NullPointerException`. This poses a grave threat, since most apps rely on some form of analytics data or advertising campaigns and thus use these services. We reported these vulnerabilities in early 2019 and most of them have been fixed starting January 2020.
(2) *Unity* is a graphics engine which allows its users to develop apps which need a more complex graphical representation,

for example, games. Therefore, it provides its own cross-compilation editor environment. Our results showed `NullPointerExceptions` in the `VRPurchaseActivity` component, a child class of the `PurchaseActivity`. We discovered, that the unpacking process of the `PurchaseActivity` was flawed and lead any app using the framework to crash if attacked. This issue was fixed in late 2019.

(3) *Adjust* can be used by companies in their Android apps to gather statistics. The SDK is used to detect how frequently apps are visited by users and how much time they spend using provided features. In the scope of this paper the automated tests detected the `AdjustReferrerReceiver` component as the reason for apps crashes in different apps. Together with some of the components detected in the Google Play Services, the crashes were provoked by sending a malformed URI in the Intent's data field. This way Android's `URLDecoder` class raised an `IllegalArgumentException` causing the app crashes. This issue was fixed in early 2019.

5 FUZZER COMPARISON

Conceptually related to our work are the generic fuzzing approaches for Android's IPC, namely ComDroid [3], Intent Fuzzer [19], the NCC group fuzzer [7] and the Fuzzinozer framework [18]. All are automated or semi-automated solutions deployed on an Android device or emulator that stress test the installed apps.

Comdroid [3] uses a semi-automated, mixed approach of static and dynamic analysis to assess intent objects and the corresponding activities or services that interact with them. It can detect potential vulnerabilities, focusing on unauthorized intent receipts and intent spoofing. Contrary to our work, we create concrete Intent objects defined through our static pre-analysis of an app. Thereby our approach is better adaptable for the frequent changes in Android's IPC and system components, as we rely on them directly to create the intents and not on a formal definition. Additionally, our approach directly yields app or component crashes, taking the verification of a detected issue further.

Sasnauskas et al. [19] build on the UI testing framework Monkey [13] and FlowDroid [1] to realize an IPC fuzzer. Similar to our work they assess the structure of intents and build their fuzzing around it. However they focus on empty intents in their initial structure assessments while we focus on static processing of the intent processing component of the app, thereby creating a bigger set of valid intents before starting the fuzzing process, i.e., mutation of our valid intent set. We argue thereby, that our approach generates far more exceptions and crashes, has a bigger scope of potential vulnerabilities and tested assumptions. This claim is supported by our evaluation which, compared to the NCC fuzzer – the improved version of the null intent fuzzer – yields far more potential vulnerabilities and bugs. Their comparison with the original null fuzzer yielded only a one percent increase in code coverage, however.

Similar to the fuzzer presented in this work Fuzzinozer [18] is a module extension for the Drozer framework [11], while the NCC goup fuzzer [7] is a standalone framework. Unlike this work's approach both do not employ static app analysis of any kind and instead rely on null intent fuzzing and randomized approaches. All exported components of an app are fuzzed with randomly generated intents, without any payload or targeted approaches. Another difference is the missing log file analysis capabilities – while the NCC group fuzzer does not support it entirely, Fuzzinozer is limited to the exception type only information, meaning the advanced Stacktrace analysis, app crash, and intent structure analysis of our work are not included. The Android IPC robustness study of Maji et al. [12], who strongly relied on a null intent fuzzer and generated sets of valid and semi-valid intents also falls under the same limitations. Their generated intents focused on the object field defined in the intent specification and assessed which, if left blank or corrupted, could lead to a `Nullpointerexception`-related crash.

A direct comparison between our implementation, the NCC group fuzzer, and Fuzzinozer can be seem in table 1. We compare the fuzzers on 10 popular apps from our dataset. The results show, that our approach generated more exceptions, found more components vulnerable and yielded more crashes in these components. Moreover, as supported by the generated exceptions from figure 2, we are able to detect far more exception types than `Nullpointerexceptions`, as would be the case for a null intent-based fuzzers.

6 DISCUSSION

Designing the Intent handling process of components is one of the key challenges for robust apps. The main reason for this is that Intents lack a formal schema for their payloads, and the the data sent within an Intent is not transmitted in a fully type-safe manner. Further, there is no specification which key-value mappings the map must contain and which are optional. While Android's `Intent` class provides helper methods to extract those values and convert it into a certain datatype, these methods do not enforce the requested mapping or its data type to exist. This means there is no liability between the sender and receiver of an Intent. A certain level of mutual trust between the two is expected as they have to agree on the format of the Intents' payloads they send among each other by themselves.

One of the easiest approaches developers can take, is to make Intent processing more stable by establishing default exception handling as well as constraint and type checking. A possible way to achieve this is by implementing sub-classes for received Intents. This has two major advantages: Fist, values being expected to be present in an Intent's extra field can be mapped to fields defined in the subclass explicitly. Secondly, establishing default values for fields is much easier. If one of the expected values is not present in the payload of an incoming Intent, the component can fall back to a default values thus creating a safe error state instead of a crash. As a general approach, if the component is an activity which has a graphical user interface attached to it, default values can be used to propagate the error to the user graphically.

To perform extended input validation, it is necessary to have a precisely defined format by which the data is sent. For Intents there is a large semantic gap between the payload of the extra field, which is formatted implicitly, and what Java's object and type system can express explicitly. For example, it is not possible to enforce a value to be not null and for number types it is not possible to clamp values into certain ranges other than the number of bits used by the Android Runtime to express the type. One possibility to close

Package	Our Approach		NCC Fuzzer [7]		Fuzzinozer [18]	
	vuln. comp.	app crashes	vuln. comp.	app crashes	vuln. comp.	app crashes
com.picsart.studio	205	284	2	4	2	10
com.google.android.apps.docs.editors.sheets	11	67	1	1	0	0
com.audible.application	11	151	0	0	0	0
com.spotify.music	9	238	0	0	0	0
com.soundcloud.android	8	19	1	1	2	5
com.apple.android.music	26	14	1	1	1	1
org.telegram.messenger	13	20	1	2	3	3
com.facebook.katana	5	22	2	2	2	5
de.telekom.mail	5	9	1	1	2	3
com.google.android.apps.docs.editors.docs	4	4	1	2	0	0

Table 1: Our fuzzer in comparison to the NCC Group fuzzer and Fuzzinozer for ten popular Google Play apps. For each app the number of vulnerable app components and the number of total app crashes is shown. If two distinct intents crash the same app in the same exact point, we still count it as one.

this gap is by establishing a domain specific language (DSL) for the payload of Intents. Similar techniques were used by Remote Procedure Call (RPC) systems, in the form of interface definition languages (IDL), since their introduction by Birell et al. [2].

7 CONCLUSION

The framework presented in this paper was implemented as an extension for the drozer framework and subsequently used to fuzz test 1488 apps with an approach combining static pre-analysis and dynamic fuzz testing. We focused on the overall stability of exported components exposed by apps and determine common exception types and design flaws causing apps or components to crash. We also discovered vulnerable third party libraries shared by many apps, i.e., the Google Services Framework.

Automated fuzzing approaches for Android apps could help immensely by providing an automated portion of security testing for the growing number of Android apps. While the Google PlayStore alone hosts over 2.87 million apps [20], the Android apps ecosystem thrives on having any additional third party app stores. Keeping in mind, that apps should be tested after each version release to mitigate unwanted vulnerabilities and bugs these numbers put an immense testing overhead on the app stores themselves, as well as the developers of the apps. The robustness of many apps against attacks via Android's IPC system needs to be improved through better exception handling, automated testing, and resolving some general design flaws in components and their process of extracting and using data provided by received Intents. With our template-based IPC fuzzing framework we already provide developers with the means to include automated testing into their development process.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[2] Andrew D Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.

[3] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 239–252.

[4] Mark Deutel. 2019. Analyzer. https://github.com/markdeutel/SmaliAnalyzer, accessed on 02. April 2020.

[5] Mark Deutel. 2019. Intent Fuzzer. https://github.com/markdeutel/IntentFuzzer, accessed on 02. April 2020.

[6] Mark Deutel. 2019. Log Parser. https://github.com/markdeutel/LogParser, accessed on 02. April 2020.

[7] NCC Group. 2018. Intent Fuzzer. https://www.nccgroup.trust/us/our-research/intent-fuzzer/, accessed on 07. April 2020.

[8] Chris Haseman. 2009. *Android Essentials*. Apress.

[9] Aki Helin. 2018. radamsa. https://gitlab.com/akihe/radamsa, accessed on 07. April 2020.

[10] Yiming Jing, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi. 2016. Checking intent-based communication in android with intent space analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 735–746.

[11] MRW Labs. 2018. Drozer. https://labs.mwrinfosecurity.com/tools/drozer/, accessed on 07. April 2020.

[12] Amiya K Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. 2012. An empirical study of the robustness of inter-component communication in Android. In *Dependable systems and networks (DSN), 2012 42nd annual IEEE/IFIP international conference on*. IEEE, 1–12.

[13] Google Developer Manual. 2014. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey, accessed on 07. April 2020.

[14] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[15] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Technical Report. Technical report.

[16] Inc. PalmSource. 2005. OpenBinder. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/, accessed on 07. April 2020.

[17] Pragati Ogal Rai. 2013. *Android Application Security Essentials*. Packt Publishing Ltd.

[18] Razvan Ionescu and Stefania Popescu. 2015. Android Intent Fuzzing Module for Drozer. https://events.ccc.de/congress/2015/wiki/images/8/8d/Ccc_pdf_fuzzinozer.pdf, accessed on 07. April 2020.

[19] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 1–5.

[20] Inc. Statista. 2019. Number of Google Play Store Apps. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, accessed on 09. April 2020.

[21] Jice Wang and Hongqi Wu. 2018. Android Inter-App Communication Threats, Solutions, and Challenges. *arXiv preprint arXiv:1803.05039* (2018).

[22] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 68.

[23] Michal Zalewski. 2017. AFL Fuzzer: american fuzzy loop. https://github.com/google/AFL, accessed on 07. April 2020.