Security Testing of Second Order Permission Re-delegation Vulnerabilities in Android Apps

Biniam Fisseha Demissie
Fondazione Bruno Kessler
Trento, Italy
demissie@fbk.eu

Mariano Ceccato
University of Verona
Verona, Italy
mariano.ceccato@univr.it

ABSTRACT

In Android, inter-app communication is a cornerstone feature where apps exchange special messages called Intents in order to integrate with each other and deliver a rich end-user experience. In particular, in case an app is granted special permission, it can dispatch privileged Intents to request sensitive tasks to system components.

However, a malicious app might hijack a defective privileged app and exploit it as a proxy, to forward attacking Intents to system components. We call this threat "Second Order Permission Re-delegation" vulnerability.

In this paper, we present (i) a detailed description of this novel vulnerability and (ii) our approach based on static analysis and automated test cases generation to detect (and document) instances of this vulnerability. We empirically evaluated our approach on a large set of top Google Play apps. Results suggest that this novel vulnerability is neglected by state of the art, but that it is common even among popular apps. In fact, our approach found 27 real vulnerabilities with fast analysis time, while a state-of-the-art static analysis tool could find none of them.

CCS CONCEPTS

• **Security and privacy** → **Mobile and wireless security**; **Software security engineering**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Security testing, static analysis, fuzzing, vulnerability detection

ACM Reference Format:
Biniam Fisseha Demissie and Mariano Ceccato. 2020. Security Testing of Second Order Permission Re-delegation Vulnerabilities in Android Apps. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387905.3388592

1 INTRODUCTION

The proliferation of Android devices — from smartphones to smart TVs — opened good business opportunities even for novice developers to create and easily distribute applications (herein apps) using the centralized Android market, called Play Store, targeting millions of end-users and devices.

Unfortunately, under the high time-to-market pressure of the app ecosystem, apps are often released without proper testing and, possibly, including implementation defects. Furthermore, when apps are granted special permissions, defects could threaten the security and confidentiality of end-users data.

Felt et al. [14] defined the Permission Re-delegation vulnerability as shown in the example in Figure 1-a. The scenario to exploit this vulnerability involves *two* apps; a malicious app requests a task to a vulnerable communication app, by sending a crafted Intent message. When requested, the communication app completes the tasks by executing the method `sendTextMessage` (that is protected by the `SEND_SMS` permission) to send an SMS message. Even if *Malicious app* misses the permission to do so, it was able to eventually send an SMS, because *Communication app* is vulnerable and it re-delegated its permission to the malicious app to perform this privileged operation. SMS is a typical target of malware, because it could be used to change network operator configuration or to subscribe to premium services [23, 26, 27].

Existing literature focuses on elaborating solutions to identify permission re-delegation vulnerabilities [6, 7, 10, 20, 32]. Candidate instances of this vulnerability are detected whenever an app-under-analysis performs a privileged operation when receiving messages from another (potentially malicious) app.

According to Felt et al. [14], a possible fix consists in patching the communication app, and making it check for the permissions of the requesting app before performing privileged operations (e.g., sending an SMS). In case the requesting app misses the needed permission, the request might be discarded or the end-user might be prompted to make a decision.

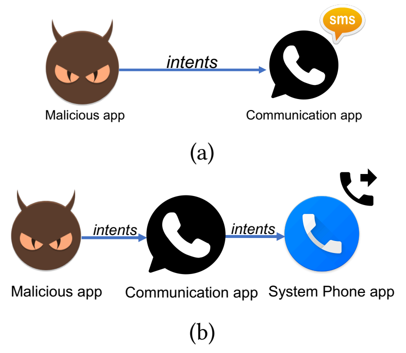


Figure 1: Example of First Order Permission Re-delegation (a) and Second Order Permission Re-delegation (b) vulnerabilities.

MOBILESoft '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7959-5/20/05…$15.00
https://doi.org/10.1145/3387905.3388592

We propose a variant of this threat model, that involves a different attack scenario, i.e., with *three* apps. This scenario is shown in Figure 1-b. It includes an attacker app with no permission, a vulnerable app with a special permission and a system app. In the example, the attacker app (i.e., *Malicious app*) sends a specially crafted Intent message to a vulnerable app (i.e., *Communication app*). Instead of performing the privileged action and executing the privileged method, the vulnerable app in turn requests the system app (i.e., system *Phone* app) to complete the privileged action — making a phone call in this case.

The attack consists in letting the attacking app initiate a phone call, even if this app is not granted the `CALL_PHONE` permission, that would have been required to do so. The attacking app exploits the vulnerable *Communication app* as a proxy to reach the system app, that eventually initiates the phone call. We should note that even if the system app adopted the patch suggested by Felt et al., the second order attack is still a threat, because the system app would anyway recognize that the last Intent is coming from the *Communication app*, that holds the required permission. Therefore, according to Felt et al., the task should not be blocked.

The difference between second order (Figure 1-b) and the first order (Figure 1-a) vulnerabilities, i.e., the vulnerable app sends an Intent message to the system instead of executing a privileged method, makes the second variant difficult to detect statically:

- The vulnerability in the *proxy* app cannot be detected by the existing automated approaches (e.g., [6, 7, 10, 20, 32]), because the vulnerability does not involve executions of permission-protected API methods, but a primitive to send inter-component (or inter-app) Intent messages;
- Inter-component communication is a very common behavior in Android apps and cannot be considered sensitive in general. If all messages sent by the proxy app were considered suspicious, an overwhelmingly high number of security notices would have to be reported, thus, too many false alarms for such an approach to be effective. Therefore, a more fine-grained analysis is required;
- It is hard for the *system* app to block the attack, because the request is coming from a benign (although vulnerable) *proxy* app that is granted special permissions.

Cases of these vulnerabilities have been reported on real apps[1] and confirmed by mainstream vendors[2].

Although multi-app analysis techniques are available in literature in order to analyze colluding apps [6, 12, 24, 29], they would not be effective in detecting Second Order Permission Re-delegation. These techniques, in fact, are based on the assumption that the colluding apps collaborate to perform malicious operations. For instance, a first app collects user data and the second app leaks these data. These approaches are mainly focused on exposing deliberate collusion for privacy leaks rather than permission re-delegation vulnerabilities due to programming mistakes.

The contributions of this paper are manifold:

- We present the *Second Order Permission Re-delegation* vulnerability and we explain why existing multi-app vulnerability analysis techniques are ineffective in detecting it;
- We propose a novel approach based on static analysis to detect instances of this vulnerability and on automated test case generation to synthesize execution scenarios that document how this vulnerability can be potentially exploited;
- We empirically assess the overall approach on more than 10K real world top apps, showing that this vulnerability affects real-world apps and that our approach is effective in detecting and testing it;
- We empirically compare our approach with a prominent existing static analysis tool, showing that this novel vulnerability is neglected by state-of-the-art automation.

The paper is organized as follows. After covering the background on Android permissions and messaging (Section 2), the new threat model is presented in Section 3. Then, Section 4 presents our novel approach to statically detect instances of vulnerability and to automatically generate test cases for them. In Section 5, a detailed empirical validation is presented. After discussing related work in Section 6, Section 7 closes the paper.

2 BACKGROUND

2.1 Application Permission

In the Android model, each app runs in a sandbox, ensuring that apps can access only their own resources and not those that belong to other apps. Furthermore, to limit the level of access, Android implements a permission based system and apps, by default, have limited permission. To access confidential informations (e.g., pictures from the camera and contacts details) or to perform sensitive tasks (e.g., recording audio and making phone calls) apps are supposed to request the corresponding special permissions that the end-user has to confirm. An app declares the permissions to request in the companion *AndroidManifest* XML file, a part of the app source code.

Figure 2 shows an example of a manifest file content. In this example, the app requests two permissions: `CALL_PHONE` to initiate phone calls and `SEND_SMS` to send text messages.

```
<activity android:name=".SIP">
    <intent-filter>
        <action android:name="android.intent.action.SENDTO" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="sms" />
        <data android:scheme="smsto" />
    </intent-filter>
</activity>
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.SEND_SMS" />
```

Figure 2: Example of manifest file, with permission requests and intent-filter.

2.2 Inter-Component Communication

Inter-component communication (ICC) is the backbone of the Android framework that allows distinct apps to cooperate. Using ICC, apps can interact by sending a special message called *Intent*. For

[1]https://www.checkmarx.com/blog/how-attackers-could-hijack-your-android-camera "How Attackers Could Hijack Your Android Camera to Spy on You"
[2]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2234

instance, when a messaging app does not support a particular document format, it could request a viewer app to display the document by sending an Intent that specifies this task along with the document to render.

An Intent contains the details of the requested task: the *Action* and the *Category* specify the operation to be performed, the *Data* usually contains a reference to a resource needed to complete the task (e.g., the document to display) and the *Extra* contains a list of key-value pairs, useful to pass additional parameters or data.

2.3 Privileged Action Strings

Intents could, in principle, specify any arbitrary string as Action. However, particular values of the Action string are reserved for sensitive tasks, and they cannot be used by all the apps. For instance, the Action string `ACTION_CALL` is a reserved Action string that an app can use to request the (device dependent) built-in Phone app to initiate a phone call. Intents with this Action strings can be issued only by those apps that are granted the `CALL_PHONE` permission, otherwise the Intent would be blocked by the system. The motivation behind this is that the corresponding Intent would trigger a sensitive task (a phone call might have a cost), so the sender app should be qualified and authorized to request such tasks.

In this paper, we use the term *privileged Action string* for those Action strings that require a special permission to be used. Thus, Intent messages that contain privileged Action strings in this paper are referred as the *privileged Intents*. Privileged Actions strings and privileged Intents are crucial in the context of Second Order Permission Re-delegation vulnerabilities.

3 THREAT MODEL AND MOTIVATION

In this section, we introduce the threat model with the help of two motivating examples, they are two apps with different vulnerabilities. These examples come from actual cases and are quite typical of vulnerable apps. The first app in Figure 3(a) contains a Permission Re-delegation vulnerability (as defined in literature [14]), that we call *First Order* vulnerability to distinguish it form the second case. The second app in Figure 3(b) contains a different security problem that we call *Second Order* permission re-delegation vulnerability.

3.1 First Order Vulnerability

The first example (in Figure 3(a)) shows two components of an app meant to send SMS. The `onClick` method in `MainActivity` is activated when the end-user clicks the *Send* button. This activity collects data filled by the end-user in the appropriate text fields. The destination number and the message to be sent (line 4 and line 5) are read from the graphical user interface and stored in an *Intent* message as the (extra) fields `num` and `msg`, respectively.

This Intent is sent to the public `Message` activity at line 6. This second activity is in charge of dispatching SMS. It reads destination and message content from the Intent (line 13 and line 14, respectively) and then sends the actual SMS message at line 15 using the `sendTextMessage` API method. To execute this API method, the app requires to be granted the special permission `SEND_SMS`.

However, this app is vulnerable to Permission Re-delegation because of a programming mistake. In fact, the `Message` activity is public, i.e., it accepts Intents not just from components within the same app (as it should be) but also from any other app installed on the same device.

Thus, an attacker app without the `SEND_SMS` permission could exploit `Message` activity to send SMS on its behalf. The attacker just needs to send an Intent with arbitrary destination number and message content to `Message` activity that, without checking the sender's permission, would complete the privileged operations and it would send the SMS.

This scenario is known as Permission Re-delegation vulnerability [14], because a vulnerable app exposes its privileged capabilities (to send SMS) to other apps, without checking their permissions. When the vulnerable app executes a permission-protected API methods (such as `sendTextMessage`) we call it *First Order* Permission Re-delegation Vulnerability. The list of permission-protected API methods are available, for instance, in Pscout [5].

3.2 Second Order Vulnerability

Figure 3(b) shows a telephony app that lets an end-user make phone calls. When the end-user clicks a button in `MainActivity`, the (already typed) number to be called is retrieved from the graphical user interface at line 4 and used to fill the Intent that is then sent to the `Call` activity.

The `Call` activity reads the number from the incoming Intent (line 11) and uses it to create a URI that is eventually used to make the phone call. To make the phone call, instead of executing a permission-protected API method (as the SMS app did), this app instead prepares a new Intent that specifies the privileged *Action* string `"ACTION_CALL"` together with the number to call (line 14). This Intent is sent to the vendor-dependent Android system Phone app, that eventually initiates the actual phone call after checking the sender's (i.e., in this case the telephony app) permissions. To use the privileged Action string `"ACTION_CALL"` in an Intent, the telephony app must be granted the `CALL_PHONE` permission. Otherwise, the telephony app is denied permission to send this Intent.

The telephony app is also vulnerable, as the `Call` activity is exposed to other apps installed on the same device. This activity does not check the sender's permission before dispatching the privileged Intent to make the call. Therefore, this app is vulnerable to Permission Re-delegation.

3.3 Considerations

The SMS app completed the privileged task within its code by executing a permission-protected *API method*. Thus, we call it *First Order* Permission Re-delegation.

The telephony app, instead, completed the privileged task by deferring the request to a system app, by sending an Intent with a privileged *Action string*. Thus, we call this second case *Second Order* Permission Re-delegation, because a third app (i.e., the system Phone app) is involved in the scenario.

First-order vulnerabilities can be statically detected, by identifying executions of privileged API methods (as done in [6–8, 10, 14, 18, 31, 32]). Second-order vulnerabilities, however, need additional analysis to be detected. In fact, sending Intents is quite common in Android apps, and most of the time it does not require permissions. Tagging all the places where Intents are sent as potentially vulnerable would result in a huge number of false positives. Precise

```java
public class MainActivity extends Activity {
 public void onClick(View v) {
  Intent intent = new Intent(this, Message.class);
  intent.putExtra("num", gui.getNumber());
  intent.putExtra("msg", gui.getMessage());
  startActivity(intent);
 }
}
```

```java
// public component
public class Message extends Activity {
 public void onCreate(Bundle saved) {
  Intent i = getIntent();
  String dest = i.getExtra("num");
  String text = i.getExtra("msg");
  smsManager.sendTextMessage(dest, null, text, null,
      null);
 }
}
```

(a)

```java
public class MainActivity extends Activity {
 public void onClick(View v) {
  Intent intent = new Intent(this, Call.class);
  intent.putExtra("num", gui.getNumber());
  startActivity(intent);
 }
}
```

```java
// public component
public class Call extends Activity {
 public void onCreate(Bundle saved) {
  Intent i = getIntent();
  String dest = i.getExtra("num");
  Uri uri = Uri.fromParts("tel", dest, null);
  String action = Intent.ACTION_CALL;
  Intent intent = new Intent(action, uri);
  startActivity(intent);
 }
}
```

(b)

Figure 3: Example of two apps with (a) First Order and (b) Second Order Permission Re-delegation vulnerabilities.

string analysis is required to identify what *Action string* is used in the Intent, to distinguish regular Intents from privileged Intents.

Existing compositional analysis techniques for detecting colluding apps (such as [6, 12, 24, 29]) are expected not to be effective in detecting Second Order Permission Re-delegation for the following reasons:

- **API vs ICC as sinks**: in first order vulnerabilities, the security sensitive sinks are API methods that require a special permission to be invoked. In second order vulnerabilities, however, the sensitive sinks are the primitives that are used to send ICC messages (e.g., `startActivity()`). While an API method can be statically related to its permission, this is not the case for ICC messages, because the permission depends on the dynamic value of the Action string. ICC is very common in Android apps and in most cases it is not security sensitive, a precise analysis is thus required in order to identify which ones are sensitive.
- **Attacker app not available**: compositional analysis techniques require attacking and attacked apps to be both available at analysis time, in order to infer their possible interaction or collusion. However, a (second order) vulnerability should be detected when scanning a vulnerable app, even if the attacking app is not available at analysis time. In fact, requiring the attacker to be available at analysis time is a strong assumption that could be violated in realistic settings, when an attacking app is developed and installed later.
- **Attacked system app not present**: in Second Order Permission Re-delegation, the last step of an attack involves a system apps (e.g., the system Phone app) or a system component that completes the privileged action. In order to perform an offline composite vulnerability analysis, the system apps should be available at analysis time. However, system apps are vendor dependent and can be heavily customized — analysis result might differ depending when system-apps from different vendors are included in the bundle.

In the next sections, a brand new approach is presented that addresses the distinctive features of Second Order Permission Re-delegation, and overcomes the limitations of existing approaches in detecting this novel kind of vulnerability.

4 STATIC ANALYSIS AND TEST CASE GENERATION

4.1 Overview

The overview of our approach is shown in Figure 4. It is composed of two main phases (i) static analysis, to determine the presence of a potential vulnerability; and (ii) test case generation, to automatically generate an attack scenario that documents the security defect.

The static analysis phase takes a (compiled) Android app as an input and emits a list of candidate vulnerabilities. For each vulnerability, a list the paths in the app call-graph is also reported.

The second phase instruments the app and fuzzes it, with the objective of generating Intent messages that execute the vulnerable paths detected in the first phase. These test cases will support us during manual verification of vulnerabilities.

4.2 Static Analysis

Source-sink detection: In order to identify Second Order Permission Re-delegation vulnerabilities, we need to first identify sources and sinks in the app code.

Sources are all the entry-points of public (exported) components. We identify the public components by analyzing the *manifest* file of the app under analysis. Exported components are those components for which an *intent-filter* is defined and/or those that are explicitly marked as "exported" (i.e., the `exported` attribute of the component definition is set to `true`). For example, in Figure 2, component `SIP` is public, because it defines an intent-filter with default (public) visibility.

Subsequently, the bytecode of these exported components is analyzed to detect their entry-points, such as `onCreate()` for an Activity and `onReceive()` for a BroadcastReceiver. The lists of entry-points for all the different Android components are taken

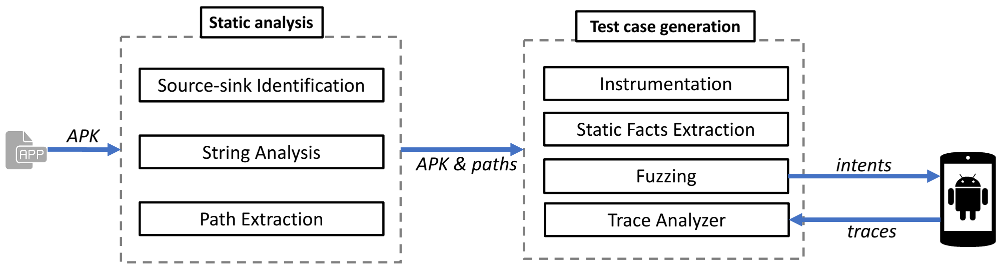


Figure 4: Overview of the proposed approach.

from the documentation of the components lifecycle (for example Activity[3]).

Sinks are invocations to ICC methods, i.e., statements that can be used to dispatch privileged Intents, such as invocations to `startActivity()`. The list of ICC methods that send Intents is available in the Android documentation[4]. However, as discussed in Section 3, not all of the invocations to ICC methods are sensitive, but only those that use privileged Action Strings. To identify what value of Action string is used in Intents, accurate string analysis is needed.

String Analysis: Sensitive sinks should be filtered to keep only those calls to ICC API methods that use privileged Action strings. For example, in Figure 3(b), line 16 of `Call` class is a sensitive sink because the Intent sent in `startActivity` uses the Action string `"ACTION_CALL"` (line 15), that requires the `CALL_PHONE` permission to succeed.

Figure 5 shows examples of how to instantiate and send privileged Intents. In Figure 5(a), the Action string is specified as an actual parameter in the call to the Intent constructor. In Figure 5(b), the Intent empty constructor is used and the Action string is added later, using the setter method `setAction`. Both of these examples use the privileged Action string `ACTION_CALL`, that requires the `CALL` permission to complete the dispatch at line 4.

In Figure 6, instead, the ICC method call at line 4 is not a sink, because the Intent created at line 2 contains the Action string `ACTION_VIEW`, that is not privileged because it requires no special permission. Thus, this ICC method invocation is not interesting and should be filtered out.

We, therefore, apply the following algorithm to compute value of Action strings:

- For each sink (i.e., a call to an ICC method), we traverse data dependence backward, until we reach the call to the constructor that instantiates the Intent object dispatched in the sink;
- From the call to this constructor, we traverse data dependence in forward direction to detect calls to `setAction`, that set the Action string in the Intent object;
- We, then, apply constant propagation to compute the value of the Action string, either in the actual parameter of `setAction` (case in Figure 5(b)) or in the actual parameter of the Intent constructor (case in Figure 5(a)).

[3]https://developer.android.com/reference/android/app/Activity
[4]https://developer.android.com/reference/android/content/Context

With the string values used as Action string, we are able to filter out regular ICC Intents that are not interesting, and we keep only a subset of sinks that are related to privileged Intents, i.e., those with privileged Action strings. Filtering is totally automated, by checking if the observed Action string matches a privileged Action String in our whitelist.

```
1  // ACTION_CALL requires CALL permission to be granted
2  Intent intent = new Intent(Intent.ACTION_CALL,[...]);
3  ...
4  startActivity(intent);  // sink b/c of tainted intent
```

(a)

```
1  Intent intent = new Intent();
2  intent.setAction(Intent.ACTION_CALL);
3  ...
4  startActivity(intent); // sink b/c of tainted intent
```

(b)

Figure 5: Variants of ICC to send Intents and make a phone call.

```
1  // ACTION_VIEW does not require any permission
2  Intent intent = new Intent(Intent.ACTION_VIEW);
3  ...
4  startActivity(intent); // not sensitive sink
```

Figure 6: ICC method call that does not require a special permission.

Path Extraction: The call graph of the app is then analyzed to identify the paths from public entry-points to the (filtered) sinks. The call graph is computed using Soot [25] with FlowDroid [4] plugin for Android. Starting from the sink nodes, the call graph is traversed backward in depth-first search manner, until a public entry-point node is reached. During this visits, visited nodes are marked, so loops in the graph are iterated at most once.

The output of this step is a list of paths in the call graph, where each path connects a public entry-point to a sink. These paths represent the vulnerabilities detected by static analysis.

4.3 Test Case Generation

Once the list of paths is available after the static analysis phase, the next step is to fuzz input values and generate test cases that make the execution traverse these paths. To this end, we first instrument the app to trace method execution and install the app on an Android emulator.

Instrumentation: The code of the app is instrumented to trace method execution and assess whether a test case was able to execute a target path. A log instruction is added before each opcode related to method call. Our log contains the information to fully reference called method, i.e. class name, method name, types of formal parameters. Execution traces will be appended to the standard Android log (with a recognizable prefix) that can be read at execution time via ADB (Android Debug Bridge).

We additionally instrument ICC method calls to log the values of Action string that are actually used at runtime when Intents are dispatched in sink statements.

Static Facts Extraction: We perform static analysis on the app bytecode to collect various information, that will be relevant to the subsequent fuzzing step:

- *Manifest*: We analyze the app *manifest* file to extract intent-filters. These filters contain detailed information about the fields expected in an Intent, i.e., Action, Category and Data. For Data, additional information can be retrieved, e.g., host or scheme. For example, from the Android Manifest shown in Figure 2, we can identify that Activity SIP is expecting an Intent message with the *Action* field set to `"SENDTO"`, *Category* field set to `"DEFAULT"` and *data* field with *scheme* set to `"sms"` or `"smsto"`. These values will be helpful to correctly generate Intent messages in the fuzzing step.
- *Code*: We analyze the app bytecode to extract all the constant strings that could possibly be used in Intent messages using the following strategies:
  1. We detect methods used to read values form incoming Intents. Method names reveal the types of Intent fields. For instance, `getStringExtra` reveals that the type of the Extra is *string*. Similar methods are used to read other types (e.g., `getIntExtra`).
  2. The actual parameters used in these methods revel expected values. For instance, when calling `get*Extra`, the first parameter is the *key* of the Extra, while the (optional) second parameter is the default value. Calls to `hasCategory` reveal the expected *Category* string of an Intent.
  3. String comparison operators on Intent fields reveal the expected values, such as in `getIntent().getAction() .equals(val)`.

Fuzzing: Based on the collected facts, the different parts of the Intent message are constructed as follows:

- *Action*: The Action String is randomly selected with uniform probability among those values that have been collected from the manifest file and the app code.
- *Category*: Similar to the Action string, the value of the Category string is randomly picked from those collected from the manifest file and from the app code.
- *Data*: The Data filed is composed of sub-fields and each of them is randomly generated based on what is specified in the manifest file. For example, in Figure 2, the manifest specifies data schemes `sms` and `smsto`. Therefore, one of these two values is randomly chosen, and a random (but valid) telephone number is appended to complete a valid Data, e.g., `sms:+1.123.555.1234`. Other random generation strategies are available for different schemes (e.g., `http`, `ftp`, `content`, `mms`).
- *Extra*: Values for this field come only from the code, because the manifest file has no entry for it. The extra is a list of key-value pairs. The *key* comes from the first actual parameter of `get*Extra`. Its *value* is generated in three ways: (i) the default value, when available in the call to the getter; (ii) a string value randomly picked among the constant string values available from the app code; (iii) a random value of the correct type (e.g., a string or an int). When a default value is available for numeric types, we generate a random number that is near the default value.

It should be noted that fuzzing is limited to Intents, i.e., test cases do not include GUI inputs, such as button clicks. In fact, in case any graphical user interaction is needed to reach the sink, it means that the end-user has the chance to approve or block the action. Instead, we are interested in testing vulnerabilities that can be exploited without the user's interaction.

Trace Analysis: After executing the test case, execution traces are collected from the Android log (logcat) via ADB and are analyzed to check if (i) the target path is executed and (ii), what Action string is used in sinks. If sinks are executed with privileged Action strings, the app is then considered vulnerable and the test case is emitted as a proof-of-concept attack.

5 EMPIRICAL VALIDATION

In this section, we evaluate the relevance and the effectiveness of our approach with some experiments. The empirical investigation is guided by the following research questions:

- **RQ$_1$**: What are the most commonly used privileged Action strings in privileged Intents?
- **RQ$_2$**: Is the proposed approach effective in detecting actual Second Order Permission Re-delegation vulnerabilities?
- **RQ$_3$**: How long does the proposed approach take to analyze an app?
- **RQ$_4$**: Is state of the art capable of detecting Second Order Permission Re-delegation vulnerabilities?

The first research question RQ$_1$ investigates how often privileged Intents are used by apps to verify if it is a prominent scenario among app developers. In fact, only when privileged Action strings are used, Second Order Permission Re-delegation vulnerabilities are possible. The second research question RQ$_2$ investigates whether the proposed approach could be useful to a developer or a security analyst to detect real security problems. Question RQ$_3$ quantifies the cost of using our approach in terms of the time taken to analyze a given app. A short analysis time is beneficial for tool adoption in a real production environment. Finally, RQ$_4$ investigates to what extent state of the art automation is effective in detecting Second Order Permission Re-delegation vulnerabilities.

5.1 Subject Apps and Experimental settings

In order to evaluate our approach, we considered apps coming from distinct sources:

- *DataSet*$_1$*: Top apps.* We collected in total 10K+ from the official Android app store, i.e., Google Play. They are collected by downloading those apps that, for each category, were listed as the top apps in 2015.
- *DataSet*$_2$*: Non-top apps.* We took 600 open source apps from F-Droid[5] that are also available on Google Play (i.e., real world open source apps). Moreover, we randomly sampled 600 closed source Google Play apps from the Androzoo [1] repository.

A time budget of 10 minutes was assigned to fuzz each candidate vulnerability detected by the static analysis. However, fuzzing would stop as soon as a test can be created that exercises the expected attack scenario.

The experiment has been conducted on a machine equipped with an Intel Core i7 2.4 GHz processor, 16 GB RAM running Apple Mac OS X 10.11. For input generation, we used an Android emulator running on the same machine.

5.2 RQ$_1$: Popularity of Privileged Actions Strings

Unfortunately, the official documentation of Android does not provide a single consolidated page that maps Intent Action strings with the permissions they need. This information is partial to some Action strings, and spread in several distinct places. Thus, we decided to collect the privileged Action strings from their actual uses in apps.

We applied our analysis, presented in Section 4, to get the list of all the Action strings used by popular apps (*DataSet*$_1$) and the Action strings documented by Pscout [5]. However, the data from Pscout is also partial. For example, the Action string `INSTALL_PACKAGE` is missing because it was introduced later in Android version 14. Moreover, the Action strings in Pscout are not labeled neither as *privileged* (that can only be sent by the system) nor as *regular* (that can also be sent by user-installed apps).

All these Action strings need to be manually filtered to keep only the privileged ones. Manual filtering consists in checking each Action string in the Android documentation to verify if any particular permission is mentioned. When the documentation was not exhaustive to label an Action string, we checked the corresponding implementation in source code of the Android project. After completing the Action string labeling process, we went back to apps in *DataSet*$_1$ and we counted how often Action strings are used in Intents. Figure 7 shows how often privileged Action strings occur in our dataset, i.e., how often they are used by developers to dispatch Intents.

As we can see, the most common Action strings are those related to capturing pictures (in 759 apps) and videos (in 207 apps). Then, requesting the user to enable Bluetooth is the third common task (152 apps). Adding app shortcuts on the home screen and removing apps are also a quite common privileged task (79 and 52 apps, respectively). Other relevant tasks are turning the device discoverable by other devices (25 apps), removing shortcut from the screen (22 apps) and setting the alarm (22 apps) followed by performing automatic phone call.

[5]https://f-droid.org/en/packages/

Considering these results, we can formulate the subsequent answer to RQ$_1$:

> Usage of privileged Action strings to request tasks to system apps/components is common in Android apps, and the most frequently occurring dangerous privileged Action strings are related to taking pictures, recording video, activating the Bluetooth, creating app shortcuts on home screen, and unistalling apps.

5.3 RQ$_2$: Detection of Vulnerable Apps

The proposed approach was applied on all the data sets (i.e., *DataSet*$_1$ and *DataSet*$_2$). Apps are subject to static analysis to detect candidate vulnerabilities. Then, candidate vulnerabilities are subject to fuzzing to see the feasibility of exploitation. 59 apps have been detected by static analysis and, therefore, were subject to fuzzing. However, we could only fuzz 47 of them, because the other 12 apps crashed or could not be installed on the testing device because of incompatibility with our experimental environment (e.g., version or architecture mismatch). For 30 apps, we could successfully generate test cases that expose the vulnerability.

On the remaining 17 apps, our approach failed in generating a test case for a variety of reasons. In some cases, apps required valid end-user credentials to run, so fuzzing stopped at the login activity. Though we filter out paths that involve GUI interaction, there were some cases that involved app-specific event-handlers for gestures that our filter did not recognize and, hence, these paths could not be filtered by our static analysis.

Even if we could fuzz 30 apps, manual inspection revealed that only 27 are actual vulnerabilities, and the remaining 3 cases are false positives.

Our experiment, therefore, resulted in 27 total vulnerabilities: 19 vulnerable apps from *DataSet*$_1$ and 8 from *DataSet*$_2$ (see Table 1). For each app (first column) the table reports the privileged Action strings used to send privileged Intents. The test cases generated by fuzzing have then been manually validated. The most common Action strings occurring in vulnerable apps is `INSTALL_SHORTCUT` with 11 vulnerable apps, then `IMAGE_CAPTURE` is used in 8 vulnerable apps, `CALL` occurs in 4 apps and `REQUEST_ENABLE` occurs in 2 vulnerabilities.

Manual investigation revealed that the 27 cases are actual vulnerabilities that our approach correctly detected. We are not able to quantify the false negatives, because this would have required to know what exactly are the vulnerable apps in our data set that our approach could have missed.

In the following, we discuss some cases, commenting the attack preconditions and the potential impact of an exploit.

CALL: This is probably the most intuitive vulnerability, the corresponding attack scenario is shown in Figure 1-b. An attacker app (that is granted no privilege) cannot send Intents with the privileged Action string `CALL` (`Intent.ACTION_CALL`). Thus, the attacker hijacks a vulnerable app with this permission to accomplish this task.

The attacker sends a regular intent to the vulnerable app. When receiving this intent, the vulnerable app sends a second intents with the `CALL` Action string, that is received by the system dialer to initiate a phone call that does not require the user confirmation.

Action string	Frequency
IMAGE_CAPTURE	759
VIDEO_CAPTURE	207
REQUEST_ENABLE	152
INSTALL_SHORTCUT	79
UNINSTALL_PACKAGE	52
REQUEST_DISCOVERABLE	25
SET_ALARM	22
UNINSTALL_SHORTCUT	22
ACTION_CALL	20
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	7
INSTALL_PACKAGE	2
SET_TIMER	1

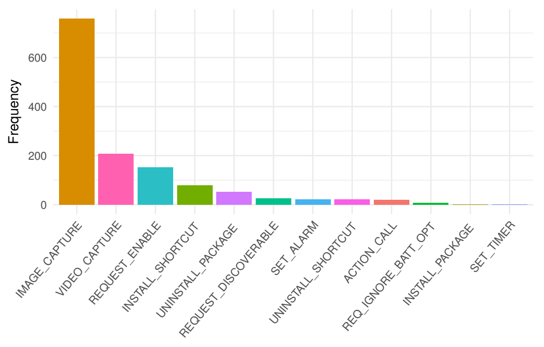


Figure 7: Privileged actions and their frequencies

App name	Action string
Dataset$_1$ (Top apps)	
cn.mstars.activity	CALL
com.mv.notas	IMAGE_CAPTURE
com.myntra.android	IMAGE_CAPTURE
com.app.app909fe1a3ad62	IMAGE_CAPTURE
com.app.app017f4c0fe778	IMAGE_CAPTURE
com.app.appd4403633f62b	IMAGE_CAPTURE
com.fm.weaponmod	IMAGE_CAPTURE
app.fastfacebook.com	IMAGE_CAPTURE
com.clearhub.wl	INSTALL_SHORTCUT
com.gtp.nextlauncher.theme.bloodysweetlove	INSTALL_SHORTCUT
com.mosoyo.wildanimals	INSTALL_SHORTCUT
com.mosoyo.watergalaxy	INSTALL_SHORTCUT
com.mosoyo.news7	INSTALL_SHORTCUT
com.mosoyo.bubbles6	INSTALL_SHORTCUT
com.moniappteam.iosnonofree.toptips4uuu	INSTALL_SHORTCUT
com.fotoable.makeup	INSTALL_SHORTCUT
com.app.all.video.downloader	INSTALL_SHORTCUT
com.ImaginationUnlimited.Poto	INSTALL_SHORTCUT
collage.instagram.b612.camera360.picsart.fotoable.instamag	INSTALL_SHORTCUT
Dataset$_2$ (Non-top apps)	
com.mvl.CasinoArizona	CALL
com.app.app0e87ec29c687	IMAGE_CAPTURE
com.yinzcam.nfl.eagles	IMAGE_CAPTURE
org.sipdroid.sipua	CALL
org.lumicall.android	CALL
org.thecongers.mtpms	REQUEST_ENABLE
a2dp.Vol	REQUEST_ENABLE
net.bluetoothviewer	REQUEST_ENABLE

Table 1: Apps vulnerable to Second Order Permission Re-delegation and their privileged Action strings.

Moreover, if the attacker is able to control the data used by the vulnerable app as a phone number to send the privileged Intent, then the attacker app can make phone calls to arbitrary phone numbers, including to premium numbers that cost actual money to the end-user.

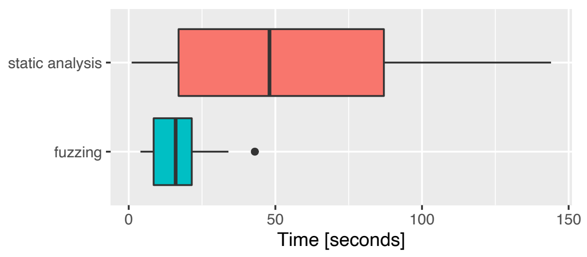
To avoid this security problem, best programming practices recommends using the Action string `DIAL`[6], that lets the end-user decide whether to complete the phone call or not.

INSTALL_SHORTCUT: If the permission related to the Action string `INSTALL_SHORTCUT` is re-delegated, an attacker can exploit this in two ways, depending on the following preconditions. (i) If the Boolean Intent extra "duplicate" is not set to false, an attacker can mount a denial-of-service attack, by filling the user's home screen with multiple icons of the same app, thus making the end-user unable to launch other apps. (ii) If the attacker can control what *Data*, *Action* and *Category* strings are used by the vulnerable apps to broadcast the privileged Intent to create the shortcut, the attacker could create a fake shortcut that resembles, for example, a banking app. The fake app could, then, redirect the end-user to a fake login screen, effectively mounting a phishing attack.

This permission should probably not be re-delegated to other apps. However, specific use cases might require this, for instance, when the developer wants to create a home screen shortcut the first time an app is launched. In these cases, shared preferences could be used to record the first creation, and checked on subsequent attempts to avoid duplication.

IMAGE_CAPTURE: This Action string can be used by a privileged app to request the camera app to capture an image and return it. Additionally, the Intent Extra field `EXTRA_OUTPUT` specifies the name of the file where to save the new picture. If an app that uses this Action string is hijacked by an attacker who also controls Intent extra `EXTRA_OUTPUT`, then the attacker might specify a path she/he can access, where the captured image will be written to. In this way, the attacking app can eventually take a picture (i.e., spying the end-user) without the need to be granted the `CAMERA` and the `WRITE_EXTERNAL_STORAGE` permissions. An attack consists in sending an Intent to the vulnerable app to request the picture and in monitoring the output directory for file changes, to steal the captured images. This is the case of a recently reported vulnerability

[6]https://developer.android.com/reference/android/content/Intent



	mean	median	sd
Static analysis	53.85	48.0	39.7
Fuzzing	16.75	16.5	9.6

Figure 8: Time (in seconds) spent during static analysis and fuzzing.

that allowed a malicious app to take photos without the required permission [7].

We can answer RQ$_2$ by stating that:

The proposed approach is effective in finding Second Order Permission Re-delegation vulnerabilities, and test cases that execute the attacks were generated for 27 vulnerable apps.

5.4 RQ$_3$: Analysis Time

To investigate RQ$_3$, we measured the amount of time spent to analyze the case study apps. To this aim, we instrumented the analysis script with the Linux `date` utility.

Figure 8 shows the box-plot of the time (in seconds) needed to perform, respectively, static analysis and fuzzing, together with descriptive statistics (mean, median and standard deviation). On average, static analysis took 54 seconds to complete on one app (including source-sink identification, string analysis and path extraction). In the worst case, the analysis took up to 140 seconds (i.e., 2 minutes and 20 seconds).

Once an app is statically detected to contain potential Second Order Permission Re-delegation vulnerabilities, we also run the fuzzer to generate inputs that execute the vulnerability.

Fuzzing was quite fast, because none of the vulnerabilities found with static analysis were tested in more than one minute. On average, fuzzing took just 16 seconds. However, the budget for fuzzing was 10 minutes and if a goal is reached within this budget, further input generation is stopped.

Based on these analysis time data, we can answer RQ$_3$ in the following way:

The cost of our approach is affordable, because static analysis of an app takes less than 1 minute in the average case, and less then 2.5 minutes in the worst case, while fuzzing took 16 seconds on average and less than one minute in the slowest case.

[7]https://www.checkmarx.com/blog/how-attackers-could-hijack-your-android-camera "How Attackers Could Hijack Your Android Camera to Spy on You"

5.5 RQ$_4$: Capabilities of the State of the Art

To tackle RQ$_4$, we consider Covert [6], a state-of-the-art static analysis tool that applies compositional analysis, whose objective is detecting if app interaction could result in a composite ICC vulnerability. An example problematic scenario detected by Covert includes two apps, the *data provider* app and the *data consumer* app. The data provider app reads sensitive end-user data and shares them via ICC to the data consumer app. The data consumer app, then, uses these data from ICC to perform some sensitive actions.

Covert supports two distinct execution modes. In the first execution mode, a set of apps is analyzed together, considering all the possible collusive pairs. In the second execution mode, Covert analyzes a single app alone, detecting if it can play the role either of data provider (sharing sensitive data via ICC) or of data consumer (using data from ICC in sensitive actions) in a potential collusion with a generic paired app. We decided to experiment with the second execution mode, because it is the most relevant to our threat model. In fact, system apps, e.g., manufacturer dependent telephony apps, might be not present in our datasets.

Covert has been run on our datasets with its default configuration. In total, Covert took 84 days to process all the apps.

Affected Sinks	**Number of report**
LOG	27,862
DATABASE_INFORMATION	9,128
NETWORK	1,442
SMS_MMS	65
AUDIO	61
NFC	20
LOCATION_INFORMATION	18
NETWORK_INFORMATION	10
CALENDAR_INFORMATION	2
ACCOUNT_SETTINGS	1
VIDEO	1

Table 2: Vulnerabilities reported by Covert, grouped by the corresponding sink privileges.

The result of the analysis is reported in Table 2. The reports contain in total 39,660 vulnerabilities. To help readability, the results are grouped by a label (first column), i.e. the affected sinks or permission needed by the sink statement. For example, if a vulnerability is potential privilege escalation to send MMS message, then the sink is `SEND_MMS`. On the other hand if the vulnerability is data leak, a possible sink is `NETWORK`. Most vulnerabilities (27,862 cases) are related to the `LOG`. The second most common class of vulnerabilities reported are related to `DATABASE_INFORMATION` (9,128 instances), either being written or leaked, followed by `NETWORK` used to communicate with remote hosts.

This list contains not only instances of permission re-delegation (privilege escalation), but also many other vulnerabilities that are out of the scope of the threat model assumed in this paper (see Section 3) and, therefore, they should be discarded. The 4,273 vulnerabilities labeled *Intent Spoofing* in the Covert report are relevant to our threat model, because they may cause privilege escalation.

Conversely, we discard those cases that are labeled *intent hijacking* (1,050 cases), because they are related to the scenario where a malicious app intercepts potentially sensitive intents, sent by a vulnerable app and meant to be received by a different app.

We also exclude the cases of information leak (34,337 cases) labeled by Covert as *Intra-app Data Leakage.* In fact, inter-app data leakage involves potentially malicious apps collaborating in order to acquire and leak sensitive data (e.g., one app reads sensitive data and another app leaks via Internet).

The 4,273 relevant reports have been subject to manual inspection to understand whether they were instances of Second Order Permission Re-delegation vulnerabilities. This task consisted in analyzing if the sink statements may send Intent messages with a privileged Action string. Manual inspection revealed that all these cases were instances of First Order Permission Re-delegation, because all the sinks were calls to Android APIs that, even if they require permissions, they were never related to inter-component communication. Thus, we can conclude that Covert detected no Second Order Permission Re-delegation vulnerability.

Considering the results obtained by Covert, we can answer RQ_4 by stating that:

The state-of-the-art for compositional analysis of Android apps is not capable of detecting Second Order Permission Re-delegation vulnerabilities. In fact, Covert detected none of the 27 vulnerabilities detected by our approach.

5.6 Limitations

The first step of our approach is based on static analysis of compiled code. Hence, the proposed approach suffers from the inherent problems of static analysis techniques, such as not precisely analyzing obfuscated code. However, while obfuscated apps are found when getting them from the official app store, when this tool is used by software developers to assess their products, the non-obfuscated version would be also available, to get more precise results.

Fuzzing is also limited, because not all the relevant attack scenarios might be generated. Though we base fuzzing on the manifest and on the code to detect *expected* Intents, more advanced input generation strategies could be more effective. For instance, search-based input generation approaches are expected to outperform fuzzing. However, fuzzing was effective in generating many proof-of-concept attacks.

Considering the fact that there is no documented list of Actions strings that require a special permission, in this work we considered the Action strings extracted from popular apps from the official app store. Though the top apps could be representative of apps in the Google Play Store, it is possible that our list misses relevant Action strings. Detailed analysis of the Android code is needed to elaborate a more complete list of the privileged Action strings, similar to what is done in Pscout [5] to identify privileged method calls.

6 RELATED WORK

Potential threats to the security of mobile apps due to inter-component (inter-app) communication have been studied with respect to information flow [9, 11, 13, 14, 17, 19, 20, 22, 28].

Felt et al. [14] defined the Permission Re-delegation vulnerability in the context of apps, together with a static analysis approach to detect it. Then, similarly to Felt et al., other approaches still limited to static analysis have been proposed [6, 7, 20, 32]. In the present paper, we extend their threat model towards the Second Order variant. Moreover, our extension includes automatic proof-of-concept attack generation.

The most related work is by Zhong et al. [32], where authors not only propose static detection of Permission Re-delegation vulnerabilities, but also test case generation. However, their investigation is limited to first order vulnerabilities, with threat scenarios that involves just two apps. Instead, we consider Second Order Permission Re-delegation, with a novel and more complex threat scenarios.

Maji et al. [21] focus inter-app messaging to generate executable scenarios that cause crashes, rather than security defects. Fuzzing was already used for security testing Android apps [15, 30]. Hay et al. [15] apply a guided fuzzing to detect several inter-app communication vulnerabilities, while Yang et al. [30] apply fuzzing based on information form intent-filters to specifically detect Permission Re-delegation vulnerabilities. Different from them, our approach includes static analysis and, only when vulnerabilities are statically detected, we apply a more complex fuzzing that is based on the results of static analysis. Additionally, we focus on a more advanced threat scenario.

Automatic testing of apps has been addressed also from the point of view of the graphical user interface [2, 3, 16], by fuzzing events in the available GUI widgets to cause exceptions, crashes, communication mistakes and violations of access permissions. However, with the focus on dependability, rather than security.

Zhang et al. [31] proposed Appsealer, to tackle permission re-delegation problems at run-time. Static data-flow analysis determines sensitive data-flows that are patched, such that the end-user is notified of a potential permission re-delegation attack and her/his authorization is needed to proceed. Lee et al. [18] presented Sealant, which extends the Android framework to deliver an approach similar to Appsealer, i.e., to monitor vulnerable inter-app communication and block them at execution time.

Both Appsealer and Sealant are based on dynamic analysis to block first order permission re-delegation, but they rely on the end-user for execution scenarios. Conversely, our approach automatically generates the execution scenarios (i.e., the test cases) and it is meant to detect second order permission re-delegation vulnerabilities.

7 CONCLUSION

Apps often access confidential end-user information and perform sensitive operations. Thus, defects that impact the end-user security should be detected and removed before apps are distributed.

In this paper, we described a peculiar vulnerability that threaten the security of Android apps, namely, Second Order Permission Re-delegation. We also proposed a novel approach based on static analysis to detect this novel kind of vulnerability and a test case generation tool that automatically creates proof-of-concept attack scenarios. Empirical results suggest that the proposed new vulnerability is common in popular apps and that our solution helps in identifying it.

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[2] D. Amalfitano, A.R. Fasolino, and P. Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on.* 252 –261. https://doi.org/10.1109/ICSTW.2011.77

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2351676.2351717

[4] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 217–228.

[6] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering* 9 (2015), 866–886.

[7] 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* (Bethesda, Maryland, USA) *(MobiSys '11)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/1999995.2000018

[8] Ting Dai, Xiaolei Li, Behnaz Hassanshahi, Roland HC Yap, and Zhenkai Liang. 2017. Roppdroid: Robust permission re-delegation prevention in android inter-component communication. *Computers & Security* 68 (2017), 98–111.

[9] Biniam Fisseha Demissie, Mariano Ceccato, and Lwin Khin Shar. 2018. AnFlo: Detecting Anomalous Sensitive Information Flows in Android Apps. In *Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. ACM.

[10] Biniam Fisseha Demissie, Davide Ghio, Mariano Ceccato, and Andrea Avancini. 2016. Identifying Android inter app communication vulnerabilities using static and dynamic analysis. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 255–266.

[11] Biniam Fisseha Demissie, Davide Ghio, Mariano Ceccato, and Andrea Avancini. 2016. Identifying Android inter app communication vulnerabilities using static and dynamic analysis. In *Proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems*. ACM, 255–266.

[12] Karim O Elish, Danfeng Yao, and Barbara G Ryder. 2015. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy.*

[13] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th Usenix Symposium on Operating Systems Design and Implementation.*

[14] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *20th Usenix Security Symposium.*

[15] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic Detection of Inter-application Communication Vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 118–128. https://doi.org/10.1145/2771783.2771800

[16] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. ACM, New York, NY, USA, 77–83. https://doi.org/10.1145/1982595.1982612

[17] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614633

[18] Youn Kyu Lee, Jae young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A SEALANT for Inter-app Security Holes in Android. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 312–323. https://doi.org/10.1109/ICSE.2017.36

[19] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. 280–291.

[20] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/2382196.2382223

[21] A.K. Maji, F.A. Arshad, S. Bagchi, and J.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.* 1 –12. https://doi.org/10.1109/DSN.2012.6263963

[22] Christopher Mann and Artem Starostin. 2012. A Framework for Static Detection of Privacy Leaks in Android Applications. In *27th Symposium on Applied Computing (SAC): Computer Security Track.* 1457–1462.

[23] Trend Micro. last accessed January 2020. First Kotlin-Developed Malicious App Signs Users Up for Premium SMS Services, https://blog.trendmicro.com/trendlabs-security-intelligence/first-kotlin-developed-malicious-app-signs-users-premium-sms-services/. https://blog.trendmicro.com/trendlabs-security-intelligence/first-kotlin-developed-malicious-app-signs-users-premium-sms-services/

[24] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. 2014. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop.* ACM, 4.

[25] Soot. 2018. Soot - A Java optimization framework, https://github.com/Sable/soot. (2018). https://sable.github.io/soot/

[26] Techspot. last accessed January 2020. New Android malware can steal data, record audio, and send SMS messages to premium services, https://www.techspot.com/news/73481-new-android-malware-can-steal-data-record-audio.html. https://www.techspot.com/news/73481-new-android-malware-can-steal-data-record-audio.html

[27] Threatpost. last accessed January 2020. Joker Android Malware Snowballs on Google Play, https://threatpost.com/joker-androids-malware-ramps-volume/151785/. https://threatpost.com/joker-androids-malware-ramps-volume/151785/

[28] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. AmAndroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[29] Mengwei Xu, Yun Ma, Xuanzhe Liu, Felix Xiaozhu Lin, and Yunxin Liu. 2017. Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web.* 143–152.

[30] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 2014. IntentFuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security.* ACM, 531–536.

[31] Mu Zhang and Heng Yin. 2014. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications.

[32] J. Zhong, J. Huang, and B. Liang. 2012. Android Permission Re-delegation Detection and Test Case Generation. In *2012 International Conference on Computer Science and Service System.* 871–874.