12-2020

Security analysis of permission re-delegation vulnerabilities in Android apps

Biniam Fisseha DEMISSIE

Mariano CECCATO

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Citation

DEMISSIE, Biniam Fisseha; CECCATO, Mariano; and SHAR, Lwin Khin. Security analysis of permission re-delegation vulnerabilities in Android apps. (2020). *Empirical Software Engineering*. 25, (6), 5084-5136.
Available at: https://ink.library.smu.edu.sg/sis_research/7133

Empirical Software Engineering (2020) 25:5084–5136
https://doi.org/10.1007/s10664-020-09879-8

Security analysis of permission re-delegation vulnerabilities in Android apps

Biniam Fisseha Demissie[1] · Mariano Ceccato[2] · Lwin Khin Shar[3]

Published online: 15 September 2020

Abstract

The Android platform facilitates reuse of app functionalities by allowing an app to request an action from another app through inter-process communication mechanism. This feature is one of the reasons for the popularity of Android, but it also poses security risks to the end users because malicious, unprivileged apps could exploit this feature to make privileged apps perform privileged actions on behalf of them. In this paper, we investigate the hybrid use of program analysis, genetic algorithm based test generation, natural language processing, machine learning techniques for *precise* detection of permission re-delegation vulnerabilities in Android apps. Our approach first groups a large set of benign and non-vulnerable apps into different clusters, based on their similarities in terms of functional descriptions. It then generates permission re-delegation model for each cluster, which characterizes common permission re-delegation behaviors of the apps in the cluster. Given an app under test, our approach checks whether it has permission re-delegation behaviors that deviate from the model of the cluster it belongs to. If that is the case, it generates test cases to detect the vulnerabilities. We evaluated the vulnerability detection capability of our approach based on 1,258 official apps and 20 mutated apps. Our approach achieved 81.8% recall and 100% precision. We also compared our approach with two static analysis-based approaches — *Covert* and *IccTA* — based on 595 open source apps. Our approach detected 30 vulnerable apps whereas *Covert* detected one of them and *IccTA* did not detect any. Executable proof-of-concept attacks generated by our approach were reported to the corresponding app developers.

Keywords Permission re-delegation · Android · Program analysis · Genetic algorithm · Test generation · Natural language processing · Outlier detection

1 Introduction

Nowadays, applications for smart phones (hereafter, apps) play an important role in our daily activities, from communication, social networking, shopping, fitness, media and

Communicated by: Eric Bodden

✉ Mariano Ceccato
mariano.ceccato@univr.it

Extended author information available on the last page of the article.

entertainment, to business and banking. These apps tend to process sensitive user information and also perform privileged actions, such as making phone calls and accessing privacy data (e.g., location). Hence, protecting sensitive user information and privileges is an essential security and privacy requirement for such apps. However, app markets are very competitive, often providing several apps with similar functionalities. Therefore, when a new idea becomes apparent, an app developer needs to rush before similar apps become available on the market by competitors. Since the first apps that appear on the market usually get accepted by the users and are rated better, posting an app early can help the developer gain market share. As a result, developers are usually under pressure to develop their apps as quickly as possible. They spend more time on providing rich functionalities and usability, often overlooking the security and privacy requirements of the app (Enck et al. 2011).

To prevent security and privacy issues, the Android operating system grants apps minimal privileges by default. The apps have to explicitly request additional permissions (that the end user has to acknowledge) to perform privileged actions, such as reading the GPS position, making phone calls or sending SMS. Hence, to avoid suspicion, malicious apps typically request for few (or no) privileges. On the other hand, Android apps can collaborate and delegate tasks among each other, by exchanging inter-process communication (IPC) messages. The possibility to request an action from insecurely-developed apps, gives rise to the threat of permission re-delegation vulnerabilities.

Permission re-delegation vulnerability is a type of privilege escalation problems. It may occur when a privileged app performs privileged actions upon request by a less privileged (possibly malicious) app (Felt et al. 2011). According to the top 10 mobile security risks reported by OWASP (2015), privilege escalation is among the most dangerous and common type of vulnerabilities in mobile apps.

In our previous work (Demissie et al. 2016), we applied static and dynamic taint analysis with the objective of detecting permission re-delegation vulnerabilities. However, taint analysis typically detects data dependencies between data from other apps' requests and data used in privileged actions. As a cornerstone feature in Android, requesting an action from another app may not always lead to a vulnerability. To limit false alarms, an accurate analysis approach should distinguish between intended permission re-delegation and actual permission re-delegation vulnerabilities.

For example, apps that need to initiate a phone call usually do not implement this feature because they assume this feature to be already available in the smart phone. They simply send a request for this action to the Phone app that processes such incoming requests by initiating the phone call (a privileged action). This is one typical privileged feature exposed by telephony apps to other apps. It is an intended feature and is neither a programming mistake nor a permission re-delegation vulnerability. However, a vulnerable version of the Phone app[1] also exposed another feature that could be used by other apps to wipe out phone data and perform factory reset. This second scenario is very uncommon among telephony apps and it represents a permission re-delegation vulnerability.

Taint analysis based approaches detect both of these permission re-delegation scenarios as potentially problematic, because they both involve a privileged action (phone dialing and data wiping) and inter-app action request. To accurately report only actual security problems, cases of permission re-delegation vulnerabilities must be distinguished from legitimate cases of permission re-delegation.

[1] Vulnerability in the Samsung TouchWiz phone dialer http://www.androidauthority.com/touchwiz-vulnerability-data-wipe-117800/

In this paper we propose a novel *P*ermission *RE*-delegation *V*ulnerability detection (*PREV*) framework, which seamlessly combines static analysis, natural language processing (NLP), machine learning, and genetic algorithm-based test generation techniques for precise detection of permission re-delegation vulnerabilities in Android apps.

More specifically, given a large training set of benign and non-vulnerable (denoted as *safe*) apps, we first apply NLP on their app descriptions and use clustering to create clusters of highly similar apps. Then, for each cluster, we apply static analysis to infer *permission re-delegation behaviors* of the apps in the cluster, i.e., privileged actions that may be performed upon receiving incoming requests. Based on this information, we build *permission re-delegation model* of the cluster, which characterizes common permission re-delegation behaviors of the apps in that cluster. Given an app under test (AUT for short), we first determine the cluster it belongs to, based on its app description (similar declared features); we then check whether the AUT has one or more permission re-delegation behaviors that deviate from the model of the cluster. If that is the case, each anomalous behavior is reported as a candidate permission re-delegation vulnerability. We then apply genetic algorithm to generate proof-of-concept attacks that exploit candidate vulnerabilities and confirm whether the AUT is indeed vulnerable and exploitable.

In our empirical assessment, we built permission re-delegation models based on the top 11,796 "safe" apps downloaded from the official Android app store (*Google Play*). We evaluated our approach based on 20 mutated apps and 1,258 real world apps (not from those top 11,796 apps) that are also available on Google Play store. Our approach achieved 81.8% recall and 100% precision. We also compared our approach with two static analysis-based approaches, *Covert* (Bagheri et al. 2015) and *IccTA* (Li et al. 2015), which can be used to detect permission re-delegation vulnerabilities in Android apps, based on 595 open source apps. *PREV* detected 30 vulnerable apps whereas *Covert* detected one of them and *IccTA* did not detect any. We reported our findings to the app developers.

To summarize, the main contributions of the paper are:

- *PREV*, a fully automated framework for detecting permission re-delegation vulnerabilities in Android apps, based on static analysis, natural language processing, machine learning, and genetic algorithm.
- A publicly-available implementation of *PREV*and dataset.[2]
- A large-scale empirical assessment, in which 11,796 apps were analyzed for learning the permission re-delegation models, and 1,258 real world apps were analyzed to detect permission re-delegation vulnerabilities.
- A comprehensive comparison with static analysis tools in terms of precision and recall.

The paper is structured as follows. Section 2 covers the background on Android and on genetic algorithms. Section 3 presents our attack model with a motivating example. Section 4 provides the overview of our approach, that is later presented in details. In particular, Section 5 describes the process to learn the permission re-delegation models, Section 6 explains how we detect anomalies with respect to this model and Section 7 details the test case generation step. Section 8 evaluates our approach. Section 9 discusses related work. Section 10 concludes the paper.

[2] https://biniamf.github.io/PREV/

2 Background

In this section, we present some background concepts used in the rest of the paper. More specifically, we first provide a short overview of how Android apps work (Section 2.1) and we then present the genetic algorithm (Section 2.2).

2.1 Android Design

Many apps are available on the official Android app store (called *Google Play*). However, apps are provided by various developers with different levels of trust. The Android framework has been designed with the two-fold objectives of (i) allowing the integration and collaboration of apps from different vendors but still (ii) guaranteeing a certain level of separation to enforce security. Separation among apps is achieved by modeling distinct apps as distinct principals, and each principal is assigned with its own privileges, adopting a permission system to regulate access to sensitive resources.

Apps are isolated from system resources. In order to access sensitive resources such as camera, GPS position, contact lists, apps have to explicitly request for permissions that must be authorized by the end users at installation time or later at runtime. The list of authorizations requested by an app is specified in its manifest file. Figure 1 shows a fragment of the manifest file of our running example app. In this example, the app requests for the permission *CALL_PHONE* to initiate phone calls.

The Android framework assigns apps with distinct Unix User IDs, so they run in their own private user space and memory. Best practices suggest to implement communication among apps through the IPC mechanism mediated by the Android framework.

Through IPC, apps can collaborate, integrate and complement each other. For instance, an app that is able to make a phone call can accept action requests so that other apps do not need to re-implement this feature. An app can delegate a specific task to another app, without actually knowing which apps are available in the current device to accomplish that task. Different users might have different installed apps that are able to make phone calls, but the requester app does not need to know which one to contact. For the requester, it is enough that the delegated app is able to make phone calls. The requester app just needs to specify what should be done (and with what data), and the framework will identify an app that is able to accomplish it. To request an action, apps use IPC messages called *intents*. Intents are messages that contain the description (in a specific syntax) of the operation that the requester needs to perform. An app may specify the services it intends to expose to other apps by means of the *intent filters* of the XML manifest file. The framework relies on this file to decide which app to delegate.

```xml
<activity android:name="DialerActivity">
  <intent-filter>
        <action android:name="android.intent.action.DIAL" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="tel" />
  </intent-filter>
</activity>

<uses-permission android:name="android.permission.CALL_PHONE" />
```

Fig. 1 Snippet of AndroidManifest XML file

Figure 1 shows a snippet of the manifest file of an Android dialer app that can be used to make phone calls. This app defines an activity (tag `<activity>`) — with an intent filter (tag `<intent-filter>`). Activity *DialerActivity* can be requested by other apps to initiate a phone call, for example, when a link with phone number (e.g., `href="tel:+1234"`) is clicked within a web page, the browser sends an intent containing the `DIAL` action, the `DEFAULT` category and phone number to this app.

Intents can be either *implicit* or *explicit*. *Implicit* intents just specify the action to be performed. The Android framework checks the intent content to decide the most appropriate destination app(s). That is, it checks the content of an intent against the *intent filters* (i.e., with the service definitions) that are specified in the manifest files of the apps installed in the device.

For example, when the user clicks on a phone number link in a web page, the browser generates an implicit intent with `DIAL` as the action and the scheme and number to call as the data, e.g., `tel:+39.0461.314.577`. As this intent matches the intent-filter in Fig. 1, the request is dispatched to the corresponding *DialerActivity* activity. This activity in our running example app becomes active and is displayed on the screen to initiate the phone call.

In *explicit* intents, the requester app specifies the receiver name as part of the intent. That is, the requester knows exactly which app to request the action from. Different Android users, however, may have a wide diversity of installed apps, therefore a specific app may not be available. Implicit intents, instead, work on the wide heterogeneity of device configurations.

2.2 Genetic Algorithm

Genetic algorithm is a population-based meta-heuristics technique proposed for solving optimization problems. An example of optimization problems is generating test inputs that are likely to expose specific program behaviors of interest. The genetic algorithm is inspired by natural evolution from biology (Holland 1975). It searches for an optimal solution by gradually evolving an initial population of random solutions through generations. Individuals more near to the final solution are rewarded with a higher probability of transmitting their chromosomes to future generations. Fittest solutions are combined together with the hope of generating fitter ones, until the optimal solution is found. The pseudocode of the abstract genetic algorithm is shown in Fig. 2. Initially, the algorithm generates random individuals (candidate solutions). Then, the algorithm loops through three main steps until the termination conditions (optimal solution found or timeout) are met. The steps are:

1. *AssessFitness*: this step computes the fitness of each individual solution and selects a set of fittest individuals (i.e., candidate solutions that are likely to generate the optimal solution).

```
function GA () {
  initialize population of random individuals
  while (termination conditions are not met) {
      AssessFitness()
      Crossover()
      Mutate()
   }
}
```

Fig. 2 The abstract genetic algorithm

2. *Crossover*: this step first pairs the individuals selected in the previous step. Then, it generates offspring from the pairs by swapping portions of their chromosomes.
3. *Mutate*: this step mutates the offspring generated in the previous step by applying certain mutation operators (such as flipping the bits). This breeds the next population for the next iteration.

Variants of the genetic algorithm are discussed in literature, with different implementations of these steps.

3 Motivating Example

In this section, we discuss permission re-delegation vulnerability with a motivation example.

Figure 3 shows the intended behavior of the Dialer app. When an intent is sent from the Dial-Pad Activity within the Dialer app or other apps such as the browser, a phone call is initiated and the end-user should confirm it. The Dialer app also allows the user to use the Dial-Pad Activity (an internal Activity) to change or read phone configurations such as device serial number (e.g., by typing *#06# to read the phone serial number). In this second case, since the request comes from the Dial-Pad (a component of the Dialer app), the Dialer app assumes that the end-user typed it and no further end-user confirmation is asked.

3.1 Attack Scenario

Apps that are granted with privileges should not contain permission re-delegation vulnerabilities; otherwise privileges could be the target of attacks. Less privileged apps could exploit such vulnerabilities by crafting malicious intent messages intended to make a vulnerable app misuse its permissions to leak sensitive data (e.g., GPS position or contacts), modify sensitive information (such as contacts or app private data) or perform costly operations (calls or SMS to premium numbers).

Figure 4 shows an example of attack scenario in which a permission re-delegation vulnerability in an app is exploited by a less privileged app to execute a privileged operation or API.

In this paper, we define a *privileged API* as an Android API that requires a special permission to be executed.

The scenario includes two apps: a benign but vulnerable *Dialer* app D and an *attacker* app A. Let us assume that D specifies the manifest file in Fig. 1. Among others, D is

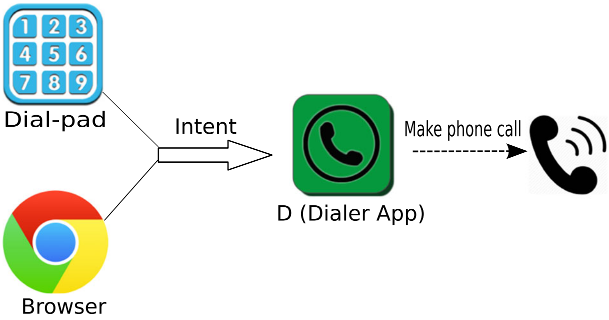


Fig. 3 Intended behavior of the Dialer app

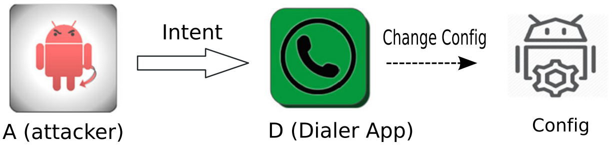


Fig. 4 An example of attack scenario

granted with the special permission `CALL_PHONE` for initiating phone call. An intent-filter is defined to allow other apps to request a phone call via this app. When *A* sends an intent message to *D*, *D* extracts the destination phone number from the message and requests a confirmation from the end-user. After the end-user confirms to call the destination number, the app initiates a call.

We define a *public entry point* of *D*, a method in the code of *D* that is executed by the Android framework when an intent is sent to *D*, e.g., the `onCreate` method in Fig. 5. Figure 5 shows the intent handling snippet of the Dialer app. The code starts by getting the intent sent to the app (Line 1). If the action in the intent is `DIAL`, the app extracts the data from the intent (Lines 2-4). The data contains the `scheme` and the phone number. If the `scheme` is “tel”, the Dialer app then extracts the number associated to this `scheme` (Lines 7-8). Then depending on the number, that is, if the number starts with * or #, the app directly performs a configuration related task (e.g., getting serial number of the phone if the number is *#06#) without asking for end-user confirmation; otherwise the app initiates a phone call.

In this example, the Dialer app has permission re-delegation vulnerability — the app makes configuration changes influenced by the data that comes from other apps. This feature is supposed to be internal, i.e., it should only be performed if the number is entered by the user using the internal dial-pad component, and no confirmation is requested from the

```
1   Intent intent = getIntent();
2   if (intent != null &&
3         intent.getAction().equals(”DIAL”)) {
4     Uri uri = intent.getData();
5
6     if (uri != null) {
7      if (”tel”.equals(uri.getScheme())) {
8       String number = uri.getSchemeSpecificPart();
9
10      if (number.startsWith(”*”) ||
11           number.startsWith(”#”)) {
12        // e.g., request USSD, MMI (wipe phone)
13        changeConfigurations(number);
14      } else {
15        makePhoneCall(number);
16     }
17    }
18  }
```

Fig. 5 Code snippet showing intent handling in Dialer app

end-user. However, by mistake, the developer exposed this capability to change configurations to other apps. As shown in Fig. 4, malicious apps could exploit this vulnerability, for example, by sending the code that wipes out the phone data.[3]

For example, when A sends an intent message to D with action `DIAL` and data `tel:*#060#`, D performs the specified task without the user interaction. In essence, D performed a privileged operation on behalf of A based on the data controlled by A without any user interaction.

Even if the attacker app A is not fully trusted, users may still install A since it requests no permission and can be assumed harmless. Even though Android treats distinct apps as distinct principals to provide separation among apps, security cannot be guaranteed when D contains such a permission re-delegation vulnerability. Exploiting this vulnerability in D, A is able to change phone state (e.g., wipe out the data from the phone or get the device serial number) without the required `MODIFY_PHONE_STATE` or `READ_PHONE_STATE` permission.

3.2 Vulnerability Preconditions

Based on the attack scenario explained above, we identify two preconditions that should be met in order to classify a case as a real permission re-delegation vulnerability.

Privileged APIs can be executed only by apps that are granted with the permission to access the sensitive resources. An attacker app that lacks the permission to access sensitive resources needs to resort to an app that holds the required access right. It needs to make the app execute privileged APIs on its behalf without the intervention of the user. Thus, the first precondition of this vulnerability is the following:

> ***Precondition PR_1: Privileged API call.*** *While performing an action requested by an intent message, the app executes a privileged API without user intervention.*

Using the example of Figs. 1 and 4, this corresponds to an app that, after receiving an intent from an attacker app, for example, formats the device by invoking the privileged API `DevicePolicyManager.wipeData(0)`, which requires the `BIND_DEVICE_ADMIN` permission.

This is a case of permission re-delegation, as described by Felt et al. (2011), because an app performs a privileged action on behalf of another app that lacks the required permission.

However, as also acknowledged by Felt et al., permission re-delegations are not always vulnerabilities; they can also be legitimate cases. Permission re-delegation is legitimate when it is an intention of the developer. In fact, in Android, inter-app communication is a cornerstone feature for app integration that involves permission re-delegation. In our running example, initiating a phone call (the method invocation `makePhoneCall()`) when requested by other app is an intended behavior of the Dialer app.

An accurate vulnerability detection approach should go beyond the mere detection of permission re-delegation and it should distinguish between legitimate permission re-delegations and permission re-delegation vulnerabilities.

Hence, going beyond Felt et al.'s threat model, we pose an additional precondition to distinguish these two cases. To consider a permission re-delegation behavior as legitimate (as intended by the developer), it should be *similar* to what can be observed on many *similar* apps. Conversely, to consider a permission re-delegation behavior as vulnerable, it should

[3] https://www.computerworld.com/article/2489707/malware-vulnerabilities/android-bug-lets-apps-make-rogue-phone-calls.html

represent an *anomaly*, i.e., something uncommon among *similar* apps. Thus, the following precondition is defined:

Precondition PR$_2$: Anomalous permission re-delegation. *It is uncommon for other similar apps to execute that privileged API upon receiving an intent message.*

We consider an app that satisfies both of the above preconditions as vulnerable to permission re-delegation attacks. In the following sections, we propose and assess an automated approach to detect apps containing such permission re-delegation vulnerabilities.

4 Overview of the Approach

The *PREV* framework is a fully-automated approach for detecting permission re-delegation vulnerabilities in Android apps. It takes as input the Android package kit (apk for short) file and the app description. As output, it generates a vulnerability report that states if the app is vulnerable or not and provides test execution scenarios with proof-of-concept attacks so as to document the security issues and help the developer in fixing the vulnerabilities.

As shown in Fig. 6, the proposed approach consists of three major steps:

1. Model inference: this step takes a large *training set* of safe apps as input and produces permission re-delegation models as output. It contains three sub-steps:
   (a) The first sub-step applies topic modeling and clustering techniques to group those safe apps into clusters based on their similarities in terms of functional descriptions.
   (b) In the second sub-step, for each app in each cluster, static analysis is used to generate the call graph and identify the privileged APIs that can be reached from public entry-points. This provides the permission re-delegation behaviors, i.e., the privileged operations that may be performed by the apps upon receiving incoming requests via public entry points.
   (c) In the third sub-step, among the reachable privileged APIs of the apps in each cluster, we determine the common APIs and the uncommon ones. Based on this information, we learn the permission re-delegation model for each cluster, which characterizes the permission re-delegation behaviors of the safe apps in the cluster.

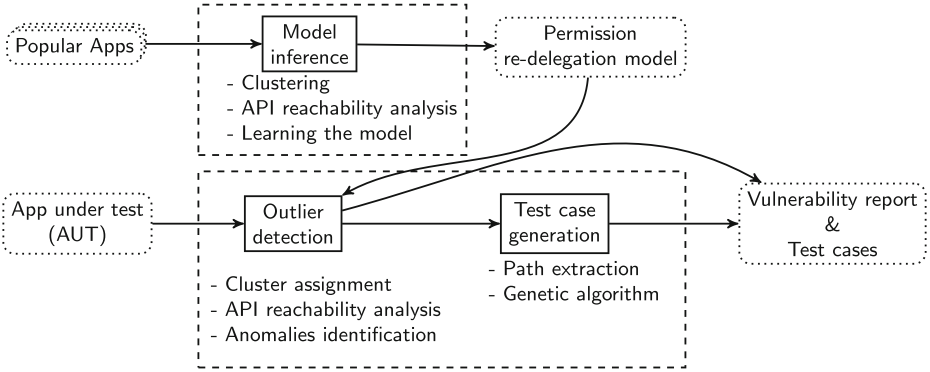


Fig. 6 Architecture of *PREV* framework

This step is performed only once before testing a given set of new apps; however, the models may need to be updated at times, for example, when new versions of safe apps become available.

2. Outlier detection: this step takes the clusters and the associated permission re-delegation models obtained in the first step and the app under test (AUT) as input. It reports anomalies as output. It contains three sub-steps:

   (a) First, it classifies the AUT into one of the clusters by using the same topic modeling technique used in the previous step and a classification technique.
   (b) Then, it proceeds to the second sub-step which applies the same API reachability analysis used in the previous step and extracts the reachable, privileged APIs in the AUT. If there is no reachable, privileged API, the procedure terminates reporting that the AUT is not vulnerable.
   (c) The third sub-step applies a classification method to identify the anomalies, which are reachable privileged APIs that are anomalous according to the permission re-delegation model. The AUT is flagged as an outlier; the anomalies are reported as *candidate* permission re-delegation vulnerabilities. If the AUT does not contain any anomaly, the procedure terminates.

3. Test case generation: this step takes the outlier AUT, the list of candidate vulnerabilities, and the call graph produced in the previous step as input. It produces proof-of-concept attacks as output. It contains two sub-steps:

   (a) It applies static analysis to extract *target paths* from the call graph — paths from public entry points to the calls to anomalous privileged APIs corresponding to candidate vulnerabilities.
   (b) Next, it applies genetic algorithm-based technique to generate test cases that exercise the target paths. This confirms that the AUT is indeed vulnerable and exploitable. It generates a detail vulnerability report containing the anomalous privileged APIs used and the exploited target paths.

These steps are described in detail in the next sections.

5 Model Inference

5.1 Clustering

In the first step of our approach, we cluster apps that can be considered benign and non-vulnerable (safe apps) based on the similarity of their app descriptions. The intuition behind is that safe apps that are similar in terms of their descriptions should exhibit common permission re-delegation behaviors, which can be considered as legitimate.

For example, it might be common for communication-related apps to send SMS messages. However, it might be very uncommon for *safe* communication-related apps to send SMS messages *when servicing requests* coming from other apps (without involving user interaction). Essentially, while this feature is largely used internally, it is rarely exposed as a service to other apps. Thus, we can establish that sending SMS on behalf of the requesting app is not a common behavior for communication-related apps. Whenever a new communication-related app is found to exhibit such behavior, it can be classified as an outlier.

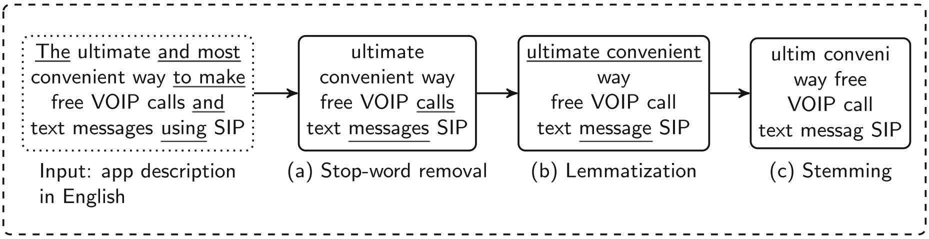

The ultimate and most convenient way to make free VOIP calls and text messages using SIP
Input: app description in English
ultimate convenient way free VOIP calls text messages SIP
(a) Stop-word removal
ultimate convenient way free VOIP call text message SIP
(b) Lemmatization
ultim conveni way free VOIP call text messag SIP
(c) Stemming

Fig. 7 App description processing (underline indicates what will be affected in the next step)

The "safe" apps that we use are those apps that (i) come from official app store (therefore, they are scrutinized and checked by the store maintainer); and (ii) are very popular (as such, their quality is acknowledged by a large group of users). We chose Google Play as the official app store. At the time we crawled the Google Play store, it provided 30 different app categories. From each category, we downloaded, on average, the top 500 apps together with their descriptions. We then discarded apps with non-English description and those with short descriptions (less than 10 words). We are then left with 11,796 apps for clusters preparation.

The fact that top apps are suggested and endorsed by the official store makes us assume that the apps are of high quality and do not contain many security problems. However, it is important to note that our approach does not absolutely assume that all the "safe" apps which are used for learning the model are completely benign and non-vulnerable. In fact, our model is robust with respect to the inclusion of a small number of malicious or vulnerable apps in the training set, because we classify a permission re-delegation behavior as vulnerable when it deviates from the cluster norm. Therefore, as long as the majority of the apps exhibit legitimate permission re-delegation behaviors, the cluster norm will only reflect those legitimate behaviors (see Section 6). On the other hand, our approach does rely on the *majority* of them being truly benign and non-vulnerable. In our empirical evaluation, we will quantify how much majority is required (see Section 8).

Our clustering step is inspired by the approach proposed by Gorla et al. (2014), with some differences in topic classification and clustering algorithm used. Specifically, we additionally apply a NLP technique called *lemmatization* for better topic classification and we use a probability-based clustering algorithm based on Expectation Maximization (EM) algorithm and cross validation method for clustering so that the number of clusters does not need to be defined a priori. This step takes safe apps as input and produces clusters of safe apps as output. It consists of three sub-steps: 1) App descriptions preprocessing; 2) Topics discovery; and 3) Apps clustering.

App Descriptions Preprocessing Our approach applies filtering, lemmatization and stemming (standard NLP techniques) to preprocess the app descriptions. The process is summarized with an example in Fig. 7. First, it filters out non-English descriptions using Google's Compact Language Detector,[4] because having one single language is necessary for clustering similar descriptions.

Second, the approach filters *stopwords* that do not contribute to topic discovery (Fig. 7a), such as "a", "after", "is", "in", "as", "very", etc.[5] Third, it applies lemmatization technique

[4] https://github.com/CLD2Owners/cld2

[5] see the list of common English stopwords at www.ranks.nl/stopwords

(Fig. 7b) using the Stanford CoreNLP lemmatizer[6] to abstract the words having similar meanings in the descriptions so that they can be analyzed as a single item. For instance, the words “car”, “truck”, “motorcycle” appearing in the descriptions can be lemmatized as “vehicle”. Last, it applies stemming (Fig. 7c) technique (Porter 1997) to transform the different forms of a word such as “travel”, “traveling”, “travels”, and “traveler” into a common base form such as “travel”.

Topics Discovery After the original app descriptions are preprocessed, the approach applies a topic modeling technique called Latent Dirichlet allocation (LDA) (Blei et al. 2003) to discover the topics in the descriptions. LDA is a generative statistical model that represents a collection of text as a mixture of topics with certain probabilities, where each word appearing in the text is attributable to one of the topics. For instance, given a preprocessed app description “travel Italy group tour include dinner lunch pizza pasta restaurant”, LDA generates the following topics with probabilities:[7] “Travel (20%)”, “Food (37%)”, “Restaurant (30%)”, “Italy (13%)”. We use the Mallet framework (McCallum 2002) to perform this step. The framework allows us to choose the number of topics to be identified by LDA. Following Gorla et al. (2014), we chose 30, the number of Google Play Store categories covered by our training and test apps.[8]

Apps Clustering After the topics are discovered, a probability-based clustering algorithm described in Witten et al. (2011) and implemented in the *Weka* tool (Hall et al. 2009), is used to group together apps based on common topics. This algorithm applies expectation maximization algorithm (Dempster et al. 1977) and cross validation method. It is as follows:

1. The number of clusters is set to 1.
2. The dataset is split randomly into 10 folds.
3. Expectation maximization is performed 10 times (in an attempt to escape local maximum).
4. The log-likelihood is averaged over all 10 results (log-likelihood is a measure of the “goodness” of the clustering).
5. If the log-likelihood has increased, increase the number of clusters by one and continues at step 2.

Expectation maximization is performed as follows: it starts with an initial guess of the cluster parameters (e.g., means and standard deviations of the clusters). It computes the probabilities for assignments of each instance to a cluster using the current parameters (expectation step). Then, using these cluster probabilities, it re-estimates the parameters (maximization step), and repeat the two steps again until the cluster parameters and cluster assignments stabilize.

The advantage of using this clustering algorithm is that it not only clusters data but also estimates the adequate number of clusters, for given data. Hence, we do not need to predefine the number of clusters. The clustering resulted in 30 clusters of similar sizes, each cluster containing between 3% and 4% of total apps. We manually sampled a few apps from a few clusters and verified that apps from the same clusters are indeed similar in terms of their functional descriptions.

[6] http://stanfordnlp.github.io/CoreNLP/

[7] To simplify the example, topics are represented by meaningful labels, which is not available in LDA.

[8] We acknowledge that choosing a different number of topics may result in different clusters. Investigating the impact of different numbers of topics is out of our scope.

We did not consider using Google Play categories as clusters. Some security analysis approaches such as Sadeghi et al. (2014) use them to avoid clustering effort. But prior result (Al-Subaihin et al. 2016) reported that clustering by common topics produces more cohesive clusters than clustering by Google Play categories because, while an app belongs to one Google Play category, an app's functional description may in fact incorporate multiple topics at once, which is a much richer information for clustering. Based on own experience (Demissie et al. 2018) and related work (Gorla et al. 2014; Avdiienko et al. 2015), categories based on topic analysis of app descriptions are more adequate than app-store categories for our security analysis purpose.

5.2 API Reachability Analysis

This step takes an app as input and produces a list of privileged APIs reachable from public entry points as output. A public entry point is an interface through which other apps, including malicious ones, can request an action via IPC. Privileged APIs are those Android APIs that require special permissions. A privileged API reachable from public entry point is a path in the call graph of the app that originates from a public entry point and that leads to a call to a privileged API.

To identify these APIs, we carry out the following tasks:

Public Entry Points Identification Public entry points are defined by *intent-filters* or the *exported* boolean attribute associated to components in the app *manifest* file, as described in Section 2.1. We model *Activities*, *Broadcast Receivers* and *Services*[9] as possible public interfaces and their corresponding lifecycle starting methods (e.g., `onCreate()` for Activities and `onReceive()` for Broadcast Receivers) as entry points. The sample manifest in Fig. 1 defines a single public interface — DialerActivity, because it defines the tag `<intent-filter>` without specifying the exported attribute (if a component specifies an intent-filter, by default the *exported* attribute is set to `true`). Our approach parses the manifest file using XOM,[10] an open source library to parse XML files. Intent-filters are extracted using XPath queries.

Privileged APIs Identification The list of privileged APIs is predefined in our configuration, which is provided in the literature (Au et al. 2012). To identify uses of these APIs in an app, we use Soot[11] to convert the Dalvik bytecode of an app into an intermediate representation called Jimple. The Jimple code is traversed to identify *invoke* statements to those APIs that match our predefined list.

Reachable Privileged APIs Identification We use *FlowDroid* (Arzt et al. 2014), which extends Soot, to generate the call graph of the app. We then run a reachability analysis algorithm (Reps et al. 1995) on the call graph to identify the privileged APIs that are reachable from public entry points.

[9] Dynamically registered broadcast receivers are not supported by our tool currently; so they are not part of our model. This represents a limitation of our current implementation.

[10] http://www.xom.nu

[11] https://sable.github.io/soot/

5.3 Learning the Model

Once the safe apps are clustered and the permission re-delegation behaviors are identified (reachable, privileged APIs), we need to learn the permission re-delegation model of each cluster.

The permission re-delegation model characterizes the permission re-delegation behaviors of the apps in the cluster, i.e., which privileged APIs are (and are not) commonly called when servicing action requests. Information about reachable APIs is stored as the matrix M, as shown in Fig. 8, with apps as rows and privileged APIs as columns. A cell in M is assigned to value 1 when the app in the row exposes the privileged API in the column when servicing action requests; otherwise it is assigned to value 0.

A column with many cells set to 1 represents a reachable API that is commonly used by many safe apps when servicing action requests; so it can be considered as a *legitimate permission re-delegation behavior*. Conversely, a column with many cells set to 0 represents a reachable API that is uncommon; so it should be considered as an *anomalous behavior*.

The notion of common/uncommon reachable APIs is captured by the frequency vector $\widetilde{m}$ that reflects the mean of the columns in M. Each element j of $\widetilde{m}$ is the mean of the j-th column of matrix M:

$$\widetilde{m}_j = \frac{1}{n}\sum_{i=1}^{n} M_{i,j}$$

Note that the lengths of frequency vectors vary across clusters. On average, $\widetilde{m}$ has 218 elements.

For example, regarding the matrix shown in Fig. 8, we have: $\widetilde{m} = [1, 0.75, 0.25, 0.25]$. The first and second elements of $\widetilde{m}$ corresponding to the APIs `openConnection()` and `connect()` have the value 1 or the value close to 1 since the APIs are frequently used while the third and the fourth elements corresponding to the APIs `sendTextMessage()` and `setWifiEnabled()` are close to the value 0 because the APIs are uncommon.

We compute a threshold called t_{comApi} to define what is common (and what is not). It is computed as the median of the values in the frequency vector, $t_{comApi} = median(\widetilde{m})$. APIs whose frequency is greater than t_{comApi} are considered as common and vice versa.

In our example, $t_{comApi} = 0.5$. The frequency of `openConnection()` is 1 and of `connect()` is 0.75; so the use of these APIs when servicing action requests is common and thus, considered as legitimate. On the other hand, the frequency of both `sendTextMessage()` and `SetWifiEnabled()` is 0.25, which is less than t_{comApi}; so they are considered as APIs uncommonly subject to permission re-delegation.

To simplify the approach, we could have set $t_{comApi} = 0$. In this way, we would identify APIs that are *never* used when servicing action requests in the given cluster as anomalous.

	openConnection()	connect()	sendTextMessage()	setWifiEnabled()
TrainingApp$_1$	1	1	1	0
TrainingApp$_2$	1	1	0	0
TrainingApp$_3$	1	1	0	0
TrainingApp$_4$	1	0	0	1

(a)

$\widetilde{m}$	1	0.75	0.25	0.25

(b)

Fig. 8 **a** Matrix M that stores information about API usage for each app in a given cluster (1=the app exposes the API, 0=otherwise), and **b** frequency vector $\widetilde{m}$ for M

However, for our approach to be robust with the inclusion of a few non-safe apps in the training set, we need a threshold larger than zero. We therefore consider those APIs that are *rarely* used when servicing action requests as uncommon by computing the threshold as explained above. In practice, even if t_{comApi} is not zero, it should still be close to zero.

Next, we compute the dispersion around the frequency vector $\widetilde{m}$ to understand how much an app should be far from this vector to be considered as an outlier. As suggested by literature (Hodge and Austin 2004), dispersion is evaluated with respect to the Euclidean distance between an app app_i and $\widetilde{m}$ with the following equation:

$$d(app_i, \widetilde{m}) = \sqrt{\sum_{j=1}^{n} (\widetilde{m}[j] - M[i, j])^2}$$

Figure 9 shows the dispersion of the distance. The upper part shows the histogram of the dispersion and the interpolating Gaussian curve. The lower part shows the boxplot.

As shown in the figure, in this example, apps have a median distance of 0.42 from $\widetilde{m}$, with very few cases with a distance larger than 0.64.

We resort to the boxplot approach proposed by Laurikkala et al. (2000) to detect outliers. The threshold called $t_{outlier}$ is computed in the same way as drawing outlier dots in boxplots:

$$t_{outlier} = Q_3 + step$$

$$step = 1.5(Q_3 - Q_1)$$

First we compute the difference between the upper quartile (75th percentile, $Q_3 = 0.64$ in the example) and the lower quartile (25th percentile, $Q_1 = 0.12$ in the example). In the example of Fig. 9, this difference is 0.52. The *step* is computed by multiplying this difference by 1.5, i.e., in the example is $step = 0.78$. Eventually, $t_{outlier}$ is computed as the sum of the upper quartile Q_3 and the step. Therefore, the threshold for the example is $t_{outlier} = 1.42$.

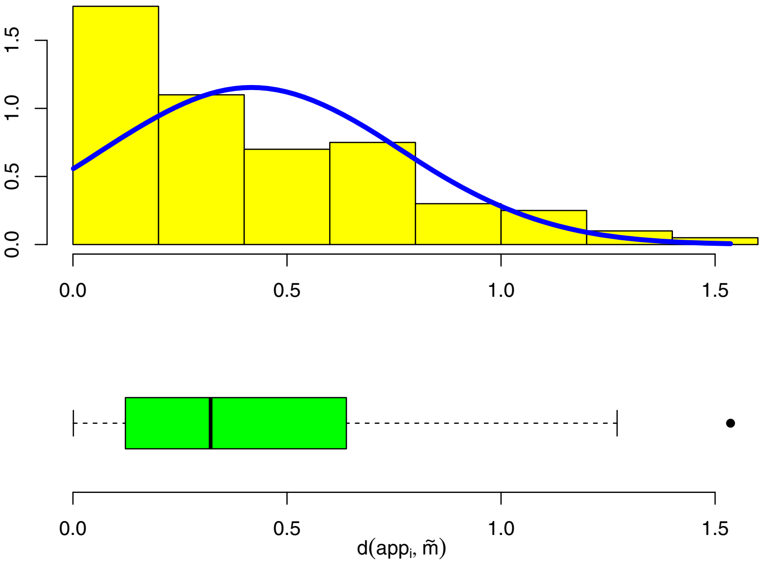


Fig. 9 Dispersion of the distance of apps from the frequency vector $\widetilde{m}$

Any app with the distance from the frequency vector $\widetilde{m}$ larger than $t_{outlier}$ is considered an outlier. For instance, the app with $distance = 1.54$ is an outlier in Fig. 9, because $1.54 > t_{outlier} = 1.42$ and hence it is represented as a dot. The frequency vector $\widetilde{m}$ and the thresholds t_{comApi} and $t_{outlier}$ represent the permission re-delegation model of this cluster.

The above process is performed for each cluster, producing a corresponding permission re-delegation model. This whole model inference step is performed only once.

The advantage of using boxplot approach is that it allows us to detect outliers in both large- and small-size clusters. Alternatively, a classification-based approach could be used to detect outliers, but classifiers typically require relatively larger data size (than a small-size cluster) to learn a robust model.

6 Outlier Detection

The second step of our approach takes as inputs the clusters and the permission re-delegation models obtained in the previous step, and the AUT. It then reports whether the AUT contains anomalous permission re-delegation behaviors. It contains three sub-steps: cluster assignment, API reachability analysis, and anomalies identification, which are explained in the following subsections.

6.1 Cluster Assignment

First of all we need to determine which permission re-delegation model to use among those available to compare the AUT against similar apps. That is, among the clusters we generated in Section 5, we need to identify the cluster the AUT belongs to. To achieve this objective, the description of the AUT is subject to the same sub-steps — app descriptions preprocessing and topics discovery — discussed in Section 5.1.

When topics and topic probabilities are computed, we use a simple, efficient classification algorithm called Naive Bayes (available in Weka (Hall et al. 2009)) to learn a classification model and assign the right cluster. The classifier is trained on the same “safe” 11,796 training apps we used for inferring the permission re-delegation models, using the topic probabilities of the apps as *features* and their clusters as *labels*. This classification model is then applied to the AUT, to identify the cluster with the most similar topic probabilities.

Figure 10 shows an example of the classification procedure. Figure 10a shows the training data, each line representing a different “safe” training app from the official store. There is a column for each topic. The value in each cell corresponds to the probability of the topic in the column given the description of the app in the row. For instance, the description of $TrainingApp_1$ is assigned to the topic “communication” with probability 0.34, the topic “health&fitness” with probability 0.09 and “games” with probability 0.52. The last column reports the cluster number assigned to this app.

Figure 10b shows the topic probabilities for the description of the AUT and the missing cluster label. Later, the Naive Bayes classifier learnt on the training data shown in Fig. 10a is used to label the AUT with the cluster whose member apps have the most similar topic probabilities with respect to the AUT.

It should be noted that clustering and training of Naive Bayes classifier is performed only once at learning time, and then it is available for classifying each AUT. Clustering and classifier training is not repeated for each AUT, so cluster assignment is expected to be fast. For cluster assignment of the AUT, we do not use the clustering algorithm used in Section 5.1

App	$Communication$	$Health\&Fitness$	...	$Games$	Cluster
$TrainingApp_1$	0.34	0.09	...	0.52	cluster20
$TrainingApp_2$	0.64	0.17	...	0.13	cluster10
$TrainingApp_3$	0.18	0.60	...	0.06	cluster26
$TrainingApp_4$	0.04	0.11	...	0.48	cluster6
...	...	...	...	...	...

(a)

AUT	0.12	0.07	...	0.78	?

(b)

Fig. 10 Classification model used for cluster assignment. **a** Topic probabilities with cluster labels, and **b** Topic probabilities for the AUT and missing cluster label

because classification would be more efficient since the group labels are already available after clustering of safe apps. Given the topic probabilities of the AUT, we simply need to classify the group it belongs to, based on the existing group labels and their associated topic probabilities. Regarding classification, we also evaluated more sophisticated classification algorithms such as Logistic Regression and Random Forest; but since the results were similar, we opted to use a simple classifier, which is Naive Bayes.

6.2 API Reachability Analysis

Like in Section 5.2, API reachability analysis is performed on the call graph, in order to identify privileged APIs that are reachable from public entry point(s). If no public entry point is found or no privileged API is reachable from public entry points, our analysis terminates here and it reports that there is no permission re-delegation vulnerability in this AUT (because permission re-delegation vulnerability arises only when an AUT executes a reachable privileged API).

6.3 Anomalies Identification

We first generate a vector x_{aut} storing the information of reachable privileged APIs in the AUT. Two examples of this vector are shown in Fig. 11, first and second line, respectively for two apps AUT$_a$ and AUT$_b$. That is, the $i-$th element of a vector is set to 1 if API$_i$ is reachable, the element is 0 otherwise. For instance, the first and second elements of $x_{aut,a}$ are set to 1 because calls to APIs `openConnection` and `connect` are reachable from public entry points in the code of AUT$_a$. Similarly, the third and fourth elements of $x_{aut,b}$ are set to 1 because calls to APIs `sendTextMessage` and `setWifiEnabled` are reachable

	openConnection()	connect()	sendTextMessage()	setWifiEnabled()	d
$x_{aut,a}$	1	1	0	0	0.43
$x_{aut,b}$	0	0	1	1	1.48
$\widetilde{m}$	1	0.75	0.25	0.25	

Fig. 11 Anomalies identification by comparing the reachable privileged APIs of the apps against those in the frequency vector

in the app code. This corresponds to computing new rows of the matrix M as discussed in Section 5.3 (see Fig. 8).

To evaluate how different AUT$_a$ is from the cluster norm, we compute the Euclidean distance d_a between $x_{aut,a}$ and the frequency vector $\widetilde{m}$ (Fig. 11). It is computed as $d_a = d(\widetilde{m}, x_{aut,a}) = 0.43$. We then compare d_a with the dispersion of permission re-delegation behaviors observed in the apps of the cluster (Fig. 9). That is, we compare $d_a = 0.43$ and the threshold $t_{outlier} = 1.42$. Since $d_a < t_{outlier}$, it is concluded that the permission re-delegation behavior of AUT$_a$ is similar to those behaviors observed in the cluster and the AUT$_a$ is flagged as normal.

Likewise, for AUT$_b$, we can compute $d_b = d(\widetilde{m}, x_{aut,b}) = 1.48$. Since $d_b > t_{outlier}$, it is concluded that the permission re-delegation behavior of AUT$_b$ is substantially different from those behaviors observed in the cluster, which is a case of *anomalous* permission re-delegation, flagging the AUT$_b$ as an outlier.

When the AUT is flagged as an outlier (as in the case of AUT$_b$), there could be two cases of anomaly: 1) the AUT *does not* expose an API that *is* commonly exposed in the cluster; or 2) the AUT *does* expose an API that *is not* commonly exposed in the cluster. Clearly, we are only interested in the second case. An outlier app might expose several privileged APIs, and the anomaly could be limited to a subset of them. Therefore, we still need to identify which privileged APIs are the anomalous ones. An API i is not commonly used for permission re-delegation in the cluster when its frequency is below the threshold t_{comApi}, i.e., $\widetilde{m}[i] \leq t_{comApi}$.

Therefore, the conditions for detecting anomalous permission re-delegation in the AUT with respect to API$_i$ are:

1. $d(\widetilde{m}, x_{aut}) > t_{outlier}$: It means that the AUT shows a permission re-delegation profile that is substantially different than the permission re-delegation profile observed on apps with similar features. So the AUT is flagged as an outlier. More conditions are required to determine what is the problematic API.
2. $\widetilde{m}[i] \leq t_{comApi}$: It means that the API$_i$ is a privileged API that is not commonly executed by the apps in this cluster when servicing action requests.
3. $x_{aut}[i] = 1$: It means that the AUT executes API$_i$ when servicing action requests coming from other apps (public entry points). In other words, the AUT exposes this privileged feature as a service to other (potentially malicious) apps.

In our running example above, AUT$_b$ is an outlier because $d_b > t_{outlier}$. The privileged APIs `sendTextMessage` and `setWifiEnabled` are not commonly executed in its cluster because $t_{comApi} = 0.5$ and $\widetilde{m}[3] = 0.25$ and $\widetilde{m}[4] = 0.25$ (Fig. 11). Therefore, the `sendTextMessage` and `setWifiEnabled` APIs satisfy the second condition. These two APIs are exposed by our outlier app AUT$_b$. As shown in Fig. 11, $x_{aut,b}[3] = x_{aut,b}[4] = 1$. Therefore, they also satisfy the third condition and are reported as anomalous privileged APIs.

Note that it is possible that vectors X_{aut} and $\widetilde{m}$ have different lengths. X_{aut} would have shorter length than $\widetilde{m}$ when the permission delegation model has APIs not observed in the AUT. In this case, we extend X_{aut} to $\widetilde{m}$'s length by adding zeros in the positions that correspond to the missing APIs. And we apply the same technique above to flag the anomalous APIs. On the other hand, when an AUT uses APIs that are never observed in the model, we flag it as an outlier and report those APIs as anomalous. We also use the above conditions 2 and 3 to flag the privileged APIs observed in the AUT but rarely observed in the model as anomalous.

7 Test Case Generation

The last step of our approach takes the outlier AUT and the list of anomalous privileged APIs as input. The objective is to generate security test cases, in the form of action requests, that execute those anomalous privileged APIs. Such test cases represent proof-of-concept attacks for permission re-delegation vulnerabilities — executable scenarios that demonstrate the presence of security defects and that document them. It contains two sub-steps: path extraction and genetic algorithm, which are explained in the following subsections.

7.1 Path Extraction

For each anomalous privileged API, the call graph of the AUT is analyzed to identify the paths from public entry points to the calls of that privileged API. Let j be a call of the anomalous privileged API. The call graph is then traversed backward in depth-first search manner starting from node j until a public entry point node is reached. During the visits, each node is marked as visited so that loops in the graph are iterated at most once. As a result, regarding each API, we obtain a list of paths.

We then filter those paths that involve an UI event. The inclusion of an UI event in a path indicates that there is a user intervention (acknowledgment) before the privileged action is taken; hence it violates our first precondition for permission re-delegation vulnerability (Section 3.2). To identify UI events, we predefine a list of UI-related callback functions such as `onClick(), onTouch()` and Android Material Design Library functions (UI-related functions). Our tool detects paths that include a call to a function from this list and discards them. The remaining paths (denoted as *target paths*) are subject to testing next.

7.2 Genetic Algorithm

This sub-step generates security test inputs that execute the targets. The security test inputs we aim to generate are in the form of intent (action request) message, which is serviced by the AUT. Our goal is to generate at least one intent message that exercises a given target path. For any anomalous privileged API that we identified above, if there is at least one target path that has been exercised, our tool reports the corresponding AUT as vulnerable. We encode the intent message generation problem as an optimization problem, to be solved by a genetic algorithm. The genetic algorithm searches for an optimal solution (serviced intent message) by gradually evolving an initial population of random individuals through generations. Individuals nearer to the final solution are rewarded with a higher probability of transmitting their genes to next generations. Fitnesses of solutions are computed using a *fitness function* and fittest solutions are combined together with the hope of generating fitter ones, until the optimum solution is found.

Individuals (also called solutions) are analogous to chromosomes in genetics. In the following, we will use the term ‘chromosome’ to refer to both individual and solution.

A chromosome is encoded as a JSON-like data structure, which contains a set of fields and their values. A chromosome contains all necessary information for generating a concrete intent message. Table 1 shows the possible fields[12] and their example values of a chromosome. Chromosomes are evolved through *crossover* and *mutation*.

[12] None of the fields is compulsory in an intent message.

Table 1 Fields of an intent message

Field	Description	Example
$action	a string representing the action to be performed	SMS, CALL, VIEW, EDIT, ...
$category	additional information about the action to perform	DEFAULT, BROWSABLE
$extra	a (possibly empty) set of key-value pairs (not specified in the manifest file)	“wifi_state”→“1”, “ volume” →10
$scheme	a value that expresses the format of the next $data field	`http` or `tel`
$data	URI that references the data to be used; e.g., the file to be opened, the number to dial or the contact to access. Depending on the $scheme, this field is composed of different subfields	`http://se.fbk.eu:80/people/profile/ceccato` `tel:+39.0461.314.577`
If $scheme is `http`, $data represents a URL with these subfields:		
$scheme	the prefix of the URI	`http`, `https`, `ftp`, `sftp`, `file`,
$host	the string corresponding to host name	`content` `se.fbk.eu`
$port	the (optional) number corresponding to the port to use	`80`
$path	The path part of a URI to locate the corresponding resource	`people/profile/ceccato`
$pathPattern	Regular expression that the path should match	`/specialdirctory.*/`
If $scheme is telephony-related, $data represents a number to call/text with these subfields:		
$scheme	the prefix of the URL	`tel`, `sms`, `smsto`, `voicemail`, `mms`, `mmsto`
$uri	the number/code to dial or to text to	`+39.0461.314.577`

The test case generation work-flow for a given target path is summarized in Fig. 12. Firstly, the *static analysis* component analyzes the Apk files of the AUT to extract the possible fields and values of intent messages that may exercise the target path, which are to be used as *seeds* for generating chromosomes (explained in the following).

Static analysis first identifies the app component that contains the target path. It then analyzes the intent-filter associated with that component in the *manifest file* and the *component code* to extract the possible fields of the intent messages that may be serviced by the component. For example, in our running example in Fig. 1, by analyzing the manifest file, we can identify that an intent message requires *$action*, *$category*, and *$data* fields so as to be serviced by the *DialerActivity* component. Note that additional fields may be identified by analyzing the component code since not all the fields are necessarily specified in the manifest file. From the component code, we also extract the *string constants* through simplified constant propagation and code scanning. Simplified constant propagation is applied to extract the values of string constants used as parameters in functions related to intents (e.g., `getIntent().getAction().equals(ACTION)`) in the corresponding component code. The technique is simplified because, for scalability reasons, we do not track the propagation of string constants through string operations such as `substring()`. Code

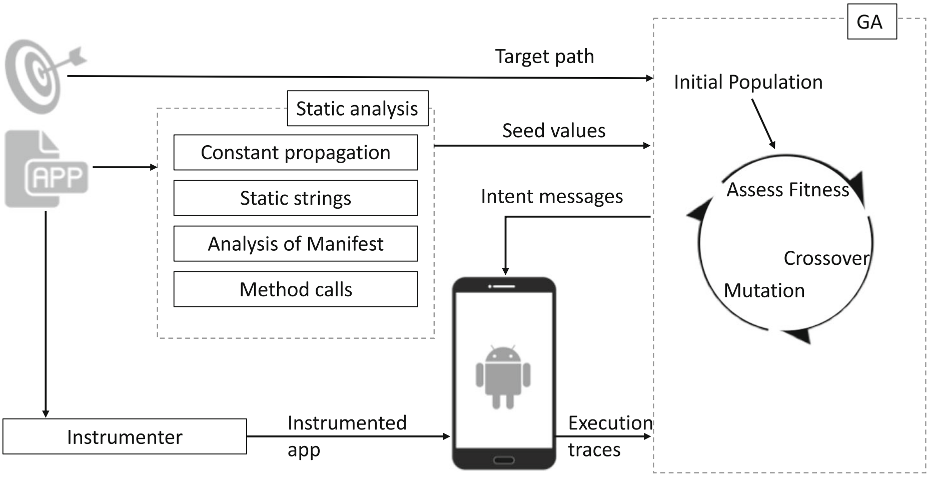


Fig. 12 Work-flow of the test case generation process

scanning is applied to extract the string constants (such as static strings) from the component code.

The following explains how the *seed values* for these fields are extracted:

$action field: its seed values are extracted from the action values specified in the manifest file (e.g. `DIAL` in Fig. 1). If no action is specified in the manifest file, its seed values are assigned with the string constants extracted from the corresponding component code (as explained above for example from `getIntent().getAction().equals(ACTION)`). Eventually, if this strategy also fails, seed values are taken from the set of all the *constant* strings that are statically available in the component,[13] in the hope of choosing a string value that is (possibly indirectly) compared to the Action when processing an Intent.

$category field: its seed values are extracted from the category values specified in the manifest file and also from the component code relevant to checking the category in intent messages (e.g., the value “Browsable” found in `getIntent().hasCategory("Browsable")`). If no such value is available, similarly to the *$action* field, string values from the constant pool of the current component are used as seed values. For instance, we used values from declarations like `String value = "Browsable";`.

$extra field: this field requires a list of key-value pairs. Since the manifest file does not specify extras, its seed values are extracted through static analysis of the component code. More specifically, static analysis is used to identify method calls that access *$extra* fields of intents and extract the keys (e.g., in `getIntent().getIntExtra("id")`, `id` is extracted as a key). Simplified constant propagation is used if the key parameter in the method call is a constant. The data type of the value is identified based on method signature (e.g., integer for `getIntExtra`). Default values for those keys are sometimes available as parameters, e.g., in `getIntent().getIntExtra("id", -1)`, -1 is a default value for `id`. If a default value is available, it is extracted as a seed value for the corresponding key. If no default value is found after static analysis of the component code, the seed values

[13] In Java, constant strings are available in the *constant pool*.

for a given key are assigned with the constants of the same data type extracted through scanning of the component code. The key is also annotated with its data type.

$scheme field: its seed value (typically only one value) is determined from the manifest file (e.g., `tel` in Fig. 1). The value of this field defines the format of *$data* field (explained next). We support 15 different *$scheme*s that are grouped into two classes: for resources such as network and contacts (e.g., `"http", "file", "content"`) and for telephony (e.g., `"tel","sms","mms"`). Custom *$scheme*s (e.g., `"fb"` for Facebook) are also supported, when they are specified in the manifest file.

$data field: this field specifies data to be used to perform the requested task. It has sub-fields depending on the *$scheme* in use, such as *$host*, *$port*, *$path*, *$uri*. Similar to the above cases, the seed values of these sub-fields are also extracted through analysis of the manifest file and the component code. The *$data* field is generated only when it is specified in the intent filter of the component.

$pathPattern field: this sub-field is usually specified in the manifest file as a regular expression (regex) consisting of the wildcards, asterisk (`*`) and a period followed by an asterisk (`.*`). The Android framework uses PatternMatcher, a simple pattern matcher that is safe to use on untrusted data and does not provide full regex support. According to the documentation,[14] an asterisk (`*`) matches a sequence of 0 to many occurrences of the immediately preceding character, while a period followed by an asterisk (`.*`) matches any sequence of 0 to many characters. The seed value for this field is, thus, generated as the shortest string accepted by the regex. For example, given a pathPattern “`/movies.*/`”, a string “`/movies/`” is generated which will be later concatenated to a URL (e.g., https://example.com/movies/).

For each of the field, we also include `NULL` value in its seed values. All the extracted fields and their seed values are then stored in a Database to be later used by the GA component to generate the chromosomes.

The *instrumenter* component in Fig. 12 instruments the AUT bytecode (based on Soot) to insert hooks at method/API invocations to trace which methods and APIs are invoked at runtime. The instrumented app is then run (in our case in the Android emulator) to process the intent messages generated by the GA component. The execution traces are logged.

The following explains how the *genetic algorithm (GA)* component works:

Initialize Population of Random Chromosomes the GA generates a population of 150 chromosomes. For each chromosome, the algorithm starts with initializing all the possible fields identified above. For the *$action* and *$category* fields, their values are randomly selected with uniform probability from their seed values extracted above. Notice that a null value may also be selected.

For the *$extra* field, we need to generate keys and values. Keys are selected randomly with uniform distribution from the ones extracted above. The value for each selected key is picked from its seed values with 70% probability or generated randomly with 30% probability. The randomly generated value is of the same data type annotated at the key. To generate a value of string data type, we give a high probability of generating a random string of up to 10 characters, based on our experience. If it is of numeric data type, we give a high probability of generating a random value close to the default value if available (i.e., added or subtracted a small value from the default value).

[14] https://developer.android.com/guide/topics/manifest/data-element

A similar algorithm is used to generate values for other fields. For example, for *$host* field, a value is picked from its seed values with 70% probability or randomly generated with 30% probability.

The same process is repeated to generate a random number of chromosomes. For each generated chromosome, a corresponding security test case in the form of an intent message that can be executed in the Android emulator is generated.

Figure 13a shows a chromosome for the running example in Fig. 1 with its *$action* and *$category* fields set to, respectively, `CALL` and `DEFAULT`. The field *$extra* contains the key `count` with the integer value `0`. The field *$scheme* is set to `tel` and the subsequent *$uri* field contains the phone number. Figure 13b shows another chromosome containing the same set of fields but with different values for some of the fields.

Figure 14 shows an ADB command that generates a concrete intent message corresponding to a chromosome.

Assess Fitness of Chromosomes this step computes the fitness of each chromosome. The objective of a security test case is to exercise the target path, from a public entry point to the

```
INTENT:
 [
  {$action: "CALL"},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 0}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
   {$uri: "+39.0461.314.577"}
   ]
  }
 ]
```

(a) *Chromosome A*

```
INTENT:
 [
  {$action: ""},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 10}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
    {$uri: ""}
   ]
  }
 ]
```

(b) *Chromosome B*

```
INTENT:
 [
  {$action: "CALL"},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 0}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
    {$uri: ""}
   ]
  }
 ]
```

(c) *Chromosome C*

```
INTENT:
 [
  {$action: ""},
  {$category: "DEFAULT"},
  {$extra:
   [
    {"count": 10}
   ]
  },
  {$data:
   [
    {$scheme: "tel"},
   {$uri: "+39.0461.314.577"}
   ]
  }
 ]
```

(d) *Chromosome D*

Fig. 13 Examples of chromosomes

```
adb shell am start
    -n com.example/.DialerActivity
    -a CALL
    -c DEFAULT
    -ei "count" 0
    -d tel://+39.0.461.314.577
```

Fig. 14 Example ADB command that sends an intent message

anomalous privileged API. Based on the execution traces logged by the instrumented code (*Instrumenter* component), the GA component determines the actual path exercised by a given test case and then uses a fitness function to compute the fitness of its corresponding chromosome.

The *fitness function* we use is similar to the approach-level introduced in Wegener et al. (2001) for the Daimler Evolutionary Testing System. However, instead of evaluating how many nodes are executed to see how far we are from the target, we compute the percentage of call edge executed. The fitness function is defined in (1). It computes the overlap between the execution that we want to achieve and the actual execution realized by the test cases as number of edges in the intersection between the executed call edges $E_{executed}$ and the call edges in the target path E_{target}. This value is then normalized in the interval [0, 1] by dividing it by the total number of the edges in the target path E_{target}.

$$fitness = \frac{\left|E_{executed} \cap E_{target}\right|}{\left|E_{target}\right|} \quad (1)$$

The larger the overlap between the target and the actual execution, the larger the $fitness$ value. When the test case executes all the edges in the target path, the $fitness$ value is one. A smaller value is obtained otherwise.

Crossover From the population of chromosomes, we use Binary Tournament algorithm to select two chromosomes based on their fitness values. The two chromosomes reaching the final of the tournament are removed from the population and subject to crossover. We pose the constraint of performing crossover only between chromosomes having the same *$scheme*. That is, if the two selected chromosomes have different *$scheme*s, they are put back into the population and the tournament is restarted.

This constraint is meant to combine only intents with compatible fields. Different schemes may imply different sub-fields of the subsequent *$data* fields. For instance, the `tel` scheme requires the *$data* field to contain only a phone number, while the `http` scheme requires the *$data* field to be composed of *$host*, *$port* and *$path* (see Table 1). Intents with the same *$scheme* ensures that *$data* are composed of compatible sub-fields, and thus can be exchanged.

We adopt a structured crossover operator that operates field-wise by crossing over fields of the same type. This is to preserve syntactic validity during evolution.

When two chromosomes A and B are selected to crossover, two new chromosomes (offspring) C and D are generated as follows:

1. chromosome A is cloned as chromosome C;
2. chromosome B is cloned as chromosome D;

Table 2 Examples of seed values for the mutation operators

Field	Seed values
$scheme	`http`, `https`, `ftp`
$extra (key=wifi_state)	`1`, `0`
$host	`se.fbk.eu`, `univr.it`
$port	`80`, `443`
$path	`people/ceccato`, `item/pen`

3. one or more fields of chromosome *C* are randomly selected, i.e., the fields and the number of fields selected for crossover could be different for different pairs of chromosomes;
4. the values of those selected fields are exchanged between chromosome *C* and *D*.[15]

To illustrate the crossover process, let us assume that *Chromosome A* and *Chromosome B* shown in Fig. 13a and b, respectively, are selected for crossover. They have the same *$scheme*, i.e., *"tel"*; therefore, crossover is allowed. Firstly, *Chromosome C* and *Chromosome D* are cloned from *Chromosome A* and *Chromosome B*, respectively. Then, assuming that the field *$uri* is randomly selected, the *$uri* values of *Chromosome C* and *Chromosome D* are swapped, resulting in two new chromosomes as shown in Fig. 13c and d.

The same process is repeated to select pairs of chromosomes from the population and crossover. This results in a new population of chromosomes, having roughly the same size as the original one (last remaining chromosomes with different *$scheme*s are discarded).

Mutation Given a new chromosome generated through crossover, the values of its fields are subject to mutation with a probability of 30%, i.e., they have 70% probability of not being mutated. Depending on the field, a different mutation operator is used, to ensure that the generated intent messages are well-formed and accepted by the app. The list of mutation operators with some examples is reported in Table 3. In these examples, we refer to the seed values shown in Table 2.

- The values of the *$scheme* field is mutated (with 30% probability) by the operator *SwitchScheme* that swaps the original value of this field with one of the seed values, selected with uniform probability. In the example of Table 3, the scheme *http* is replaced by the scheme *ftp* (available as seed in Table 2) to change the URL as shown in the corresponding line.
- The value of the *$action*, *$category* and *$pathPattern* fields are not mutated.
- For the *$extra* field, the keys are not mutated. The values of the extra are mutated with 30% probability. The mutation is performed as follows:
  - With 15% probability, the *SwitchExtraValue* operator is used to change the value of the $extra field with a seed value. In the example, the value of `wifi_state` is changed from 1 to 0, by peeking the new value from the pool of seed values for this key (see second line in Table 2);
  - With 15% probability, a *AlterExrtra*Value* operator is selected to arbitrarily change the value of the $extra value. *AlterExrtraIntValue* or *AlterExrtraStringValue* are used, depending on the type of the extra. This operator does

[15] in the case that a selected field is not present in chromosome *D*, it is ignored.

Table 3 Examples of mutation operators

Operator	Original	Mutated
SwitchScheme	http://se.fbk.eu:80/people/ceccato	ftp://se.fbk.eu:80/people/ceccato
SwitchExtraValue	wifi_state→1	wifi_state→**0**
AlterExtraIntValue	wifi_state→1	wifi_state→**5**
AlterExtraStringValue	preferred_ssid→“myhome”	preferred_ssid→“**myhom**”
	preferred_ssid→“myhome”	preferred_ssid→“myhome**X**”
	preferred_ssid→“myhome”	preferred_ssid→“my**W**ome”
SwitchHost	http://se.fbk.eu:80/people/ceccato	http://univr.it:80/people/ceccato
AlterHost	http://se.fbk.eu:80/people/ceccato	http://fbk.eu:80/people/ceccato
SwitchPort	http://se.fbk.eu:80/people/ceccato	http://se.fbk.eu:443/people/ceccato
AlterPort	http://se.fbk.eu:80/people/ceccato	http://se.fbk.eu:85/people/ceccato
SwitchPath	http://se.fbk.eu:80/people/ceccato	http://se.fbk.eu:80/item/pen
AlterPath	http://se.fbk.eu:80/people/ceccato	http://se.fbk.eu:80/people/ceccato\X

not use seed values. If the type is numeric, *AlterExrtraIntValue* mutates the value by adding or subtracting an offset. Small offsets are chosen with higher probability and the probability of larger offsets decreases exponentially. In the example the value of wifi_state is changed from 1 to 5, the value added as offset (i.e., 4) is not a seed value.

In case the type of the extra is string, the operator *AlterExrtraStringValue* is used instead. The extra value is mutated by deleting, inserting or replacing a character in the string with a random character. In the example the preferred_ssid is changed from ``myhome'', respectively, to ``myhom'', ``myhomeX'' and ``myWome''.

- For fields $host, $port and $path, the mutation operators are similar to previous cases. That is, the mutation is performed with 30% probability. With 15% probability, the field is replaced with a seed value (operators *SwitchHost*, *SwitchPort* and *SwitchPath*), and with 15% probability the value is changed regardless the available seeds (operators *AlterHost*, *AlterPort* and *AlterPath*), as shown in the corresponding examples.

Timeout The stopping criteria of the GA is set as 500 generations.

Note that the tuning parameters — the population size, the mutation probabilities, the value selection probabilities, and the stopping criteria — we used above are decided based on our preliminary assessment of the test generation algorithm, which we ran on a set of randomly selected apps. We found that higher probabilities of mutating a chromosome (>30%) and lower probabilities of selecting a value from seeded ones for mutation (<50%) usually results in the loss of good solutions. On the other hand, the test generation was not very effective when we used much lower probabilities of mutating a chromosome (e.g., 10%) and much higher probabilities of selecting seeded values (e.g., 90%). Some of the fields in intent messages, such as *$action*, *$category* *$pathPattern*, and *$key* are not mutated at all because it would only result in ill-formed intent messages that would be rejected by the AUT. Overall, this ensures that the population is evolved towards better generations.

8 Evaluation

In this section we evaluate *PREV* and compare with two state-of-the-art static analysis-based techniques — *Covert* and *IccTA* — which can detect permission re-delegation vulnerabilities.[16]

Our goal is to detect as many vulnerabilities as possible at an affordable cost. Therefore, our main evaluation criteria are precision and cost. In addition, we also investigate the recall and report the results.

More specifically, the following research questions are investigated:

- RQ_1 *(Precision)*: Is *PREV* precise at detecting permission re-delegation vulnerabilities in Android apps?
- RQ_2 *(Cost):* Is the cost (in terms of analysis time) of using our approach affordable in practice?
- RQ_3 *(Recall)*: Does *PREV* miss permission re-delegation vulnerabilities?
- RQ_4 *(Comparison):* Does *PREV* perform better than other tools that can be used to detect permission re-delegation vulnerabilities?
- RQ_5 *(Robustnesses):* Is *PREV* robust against the inclusion of anomalies in the training set?
- RQ_6 *(Threshold):* What is the impact of other threshold values on vulnerability detection accuracy of *PREV*?

Our evaluation was conducted on a machine equipped with an Intel Core i7 2.4 GHz processor, 16 GB RAM, running Apple Mac OS X 10.11. Our tool is instrumented to log the analysis time.

8.1 Subject Apps

Our subject apps include a total of 1,258 real world apps from the official Google Play store.[17] Since our approach works on compiled apps, the availability of source code is not a requirement. Nevertheless, 595 of our subject apps are open source projects, which offer us the possibility to inspect the source code, determine the correctness of vulnerability reports generated by the tools, and analyze the causes of vulnerabilities. The following explains our selection process of subject apps:

First, we obtained a list of app names from the directory of AndroZoo,[18] an app crawling research project that lists app names from many official and unofficial app repositories. We then randomly sampled the names from this list. Additional app names are also taken from a repository that collects open source Android apps, namely F-Droid.[19] Among those sampled apps, we picked those that are also available on the Google Play store, to ensure that our subject apps are real world apps.

We then filtered out those apps that are too popular (more than 1 million downloads), to increase the chance of selecting apps that are interesting for our experiment, i.e., apps

[16] We did not compare with test generation-based approaches because we could not find an adequate or available tool that might be suitable for detecting permission re-delegation vulnerabilities.

[17] Note that our tool *PREV* was trained on 11,796 official apps (see Section 5). Those apps are not considered in the selection of subject apps.

[18] https://androzoo.uni.lu/

[19] http://f-droid.org/

with a good chance of containing vulnerabilities. Popular apps, distributed by well-reputed companies, are probably already subject to intensive security review.

To be able to apply our analysis, we additionally require that app description is in English, that contains at least 10 words so that natural language processing and topic discovery can be performed.

Eventually, there remained 1,258 apps — 663 closed source and 595 open source — from the official app store together with their descriptions. For open source apps, we acquired source code to conduct manual verification later.

The list of training apps and subject apps along with the implementation of *PREV* is publicly available.[20]

8.2 Metrics

To answer our research questions, we report results in terms of these metrics:

- *Number of true positives (TP):* Number of real vulnerable apps correctly reported as vulnerable;
- *Number of false positives (FP):* Number of vulnerable apps incorrectly reported as vulnerable (false alarms);
- *Number of false negatives (FN):* Number of vulnerable apps that are missed (not reported by the tool);
- *Analysis time:* The time (measured in minutes) taken by the tool to analyze a subject app;
- *Number of contaminated apps:* Number of training apps that contain permission re-delegation vulnerabilities;
- *Threshold:* the value used to flag outlier apps

To answer RQ_1 and RQ_4, we quantify the precision of the tool based on *true positives* and *false positives* (Precision=TP/(TP+FP)). More specifically, when a tool reports a vulnerability, when source code is available, we manually inspect the part associated with the reported vulnerability. When source code is not available, we resort to the test case generated by the tool and observe the actual runtime behavior exercised by the test case. We then determine if the report is a true positive or a false positive. Note that since *Covert* and *IccTA* do not generate test cases, we evaluated them only based on open source apps so that we can verify their vulnerability reports. We answer RQ_2 by using the *analysis time* to quantify the cost of using the tool. To answer RQ_3 and RQ_4, we measure the recall of the tool based on *true positives* and *false negatives* (Recall=TP/(TP+FN)).

The challenge here is to establish the *false negatives*, we would need to thoroughly inspect the source code of the subject apps, and determine if they are vulnerable or *absolutely safe*. This would require an overwhelming effort. Therefore, instead of conducting a security review of all the subject apps to label them as safe/vulnerable, we conduct a *controlled experiment* in which we apply *two mutation operators* to inject security faults that reflect realistic permission re-delegation vulnerabilities into a set of randomly selected apps. This provides us a benchmark for evaluating the recall, where all the apps in this benchmark are vulnerable by construction.

We answer RQ_5 by including a set of contaminated apps in the training set and evaluating whether *PREV* can still detect the same permission re-delegation vulnerabilities as before.

[20] https://biniamf.github.io/PREV/

We answer RQ_6 by evaluating the impact of different threshold values on the number of vulnerabilities detected.

8.3 RQ$_1$: Precision

We ran our tool on the 1,258 subject apps. Each app under test (AUT) was subject to outlier detection and test case generation steps shown in Fig. 6.

Given the available clusters (Section 5), we map each AUT to the cluster with most similar app descriptions. We then obtain the permission re-delegation model inferred on the corresponding cluster. Next, we ran API reachability analysis on the AUT, to identify those privileged APIs that are reachable from public entry points. Out of all the 1,258 apps, 401 apps contain reachable privileged APIs.

We then performed anomalies identification, which basically checks if the identified reachable privileged APIs are common or anomalous according to the permission re-delegation model. *PREV* detected that in 324 apps, the reachable privileged APIs are common according to the permission re-delegation model and therefore, they were classified as safe. The remaining 77 apps were classified as candidate vulnerable apps because the reachable APIs in those apps are anomalous. These candidate vulnerable apps were then subject to the test case generation phase of *PREV*, to automatically generate proof-of-concept attacks. *PREV* successfully generated attacks for 30 of these apps. (Note: we also used open source apps from those 77 apps to evaluate recall in Section 8.5.)

We then face the challenge of manually analyzing the apps for which a test case is generated, to label the analysis results as true positive (real vulnerability) or false positive (false alarm). To classify a reported app as vulnerable, first we check that permission re-delegation has occurred, i.e. that a test case makes the app execute a privileged API. Then we verify whether this case of permission re-delegation is a vulnerability. To limit subjectivity in verifying this second condition we adopt these guidelines (explained in more detail later when we discuss the results):

- *Custom protocol*: the vulnerability can be triggered only with a particular message that follows an application-specific invocation protocol;
- *System intents*: the vulnerable component subscribed for system-generated events, but it fails to check whether the notified event is actually generated by the system;
- *Misuse of libraries*: the vulnerable app performs an insecure use of a library that deals with sensitive data;
- *App description*: a permission re-delegation causes the vulnerable app to perform a privileged task that is not explicitly specified as a feature in the app description.

Table 4 shows the results. The first column shows the open source apps followed by the closed source apps that are reported as vulnerable by *PREV*. The two columns (‘TP’ and ‘FP’) indicate whether the reported vulnerable app is a true positive or a false positive, respectively, based on our manual verification. Moreover, for each ‘TP’ case, the table reports what guideline has been followed to manually classify it as a real vulnerability.

Analyzing the 1,258 subject apps, *PREV* reported 30 vulnerable apps — 7 open source apps and 23 closed source apps. Manual inspection on the reported vulnerabilities revealed that, for all of them, the test cases generated by *PREV* actually reached a privileged API. Moreover, all the cases represent permission re-delegation vulnerabilities according to our guidelines. In the following we discuss some of those cases in detail to highlight the reason for their classifications.

Table 4 Vulnerability report of *PREV* on open source and closed source apps

App	TP	FP	Guideline
com.mendhak.gpslogger	✓		App Description
com.seafile.seadroid2	✓		App Description
org.ligi.ajsha	✓		App Description
org.linphone	✓		App Description
org.tigase.messenger.phone.pro	✓		App Description
org.totschnig.myexpenses	✓		App Description
org.ttrssreader	✓		App Description
bestvalleygames.turningvalley	✓		App Description
com.akgun.uknews	✓		Custom Protocol
com.appportunity.androidpreviewer	✓		App Description
com.appreka.mycoop	✓		Misuse of Library
com.appsdv.smsmefitr	✓		Misuse of Library
com.aurorasi.aurorasfa	✓		Misuse of Library
com.bimandika.Congratulationsmalonepost	✓		System Intents
com.braingen.devanagarinotepad	✓		App Description
com.dinosaur.dinosaur_vs_zombie	✓		App Description
com.fmplural.radio	✓		App Description
com.innogang.kollywoodNews	✓		App Description
com.javirurro.games.spaceshipzigzag	✓		App Description
com.josejoaquin.traductor	✓		App Description
com.magmamobile.game.SpiderSolitaire2	✓		Custom Protocol
com.netdania	✓		Custom Protocol
com.npes87184.s2tdroid	✓		App Description
com.rbsoftware.pfm.personalfinancemanager	✓		Misuse of Library
com.reverbnation.artistapp.i739749	✓		System Intents
com.softdx.qrscanner	✓		App Description
com.superfanu.bryantbulldogrewards	✓		System Intents
com.vent	✓		Misuse of Library
lv.delfi.ru	✓		Custom Protocol
piproduction.frankthejew	✓		Custom Protocol

Custom protocol app components may use custom invocation protocols such as private *actions* only known to the app (e.g., not specified in the intent-filter) or use specific values in custom *extra* parameters. Private action is an action that is private to the AUT, because (i) the action name is not mentioned in the app intent-filter, where the call protocol is exposed to other apps; and (ii) the action name has the same prefix as the app package name, e.g., the action `com.example.TestApp.TEST_ACTION` for the app `com.example.TestApp`; it is not from an included library or from the Android framework. Therefore, this action is likely for internal use, i.e., only for components of the AUT or only for apps developed by the same developers who know the internal details of the app. It is highly unlikely that this component intends to accept action requests from other external apps. Therefore, when there is a permission re-delegation scenario in which intent messages can invoke such components, we believe that this is a developer's mistake or she/he adopts a security-by-obscurity approach. This is a vulnerability because it can be uncovered by an approach like ours. This guideline was applied to classify 5 vulnerable apps such as *com.netdania* and *piproduction.frankthejew*.

System Intents apps may subscribe for notification of system events via intent filters — events that Android platform generates. For example, the app *com.superfanu.bryantbulldogrewards* subscribed to be notified when the boot is complete, i.e., it has registered an intent-filter to receive an intent with `ACTION_BOOT_COMPLETED` action, which Android platform generates on completing the boot. However, the component of this app does not validate that this notification was actually sent by the system. It blindly assumes that any intent sent to this component is from the system and processes as such. Therefore, when the intent filter specifies a system action but the app code does not validate intent data, we assume that it is a programming mistake and we classify the case as a permission re-delegation vulnerability. This guideline was applied to classify 3 vulnerable apps.

Misuse of Libraries apps may be granted with special permissions to use libraries that deal with sensitive data; but they are expected to adhere to the security policies specified in the library documentations. Therefore, when an app uses a special library in a way that violates the library security policies, we assume that it is a programming mistake and classify the case as a permission re-delegation vulnerability. For instance, the apps *com.appsdv.smsmefitr* and *com.aurorasi.aurorasfa* use the *OneSignal* and *Google Analytics* libraries, respectively. However, the app components using those libraries process broadcasts without verifying that the intent, specified with the protected broadcast action `ACTION_MY_PACKAGE_REPLACED`, is actually sent by the *PackageManager* (the system). System actions such as `ACTION_MY_PACKAGE_REPLACED` are actions that can only be set by the system when sending an Intent. If an app registers to receive a broadcast Intent with such actions, the Android system guarantees that the Intent is sent only by the system. However, apps still have to verify if the Intent they receive is actually sent by the system by checking if the action matches exactly the one that they registered to receive. If an app fails to verify and simply assumes that the Intent came from the system, then the app is potentially vulnerable. This is because a malicious app may send an Intent directly to the vulnerable app component with an arbitrary action and trick the app into performing a privileged action. This guideline was applied to classify 5 vulnerable apps.

App Description if none of the previous consideration applies, we resort to the features described in the app descriptions to understand if permission re-delegation is *intentional*. From the descriptions, one can find out about the primary features of an app (e.g., accessing the camera by a photography app). It might be the intention of the developer to expose these primary features to other apps (and let other apps request this app to take pictures on their behalves). However, when features that require privileged permission but not described in the description are exposed to other apps, we assume that it is not the developer's intention and classify the case as a permission re-delegation vulnerability. For instance the app *com.appportunity.androidpreviewer* is described as a gallery of apps, to help developers keep track of their apps. However, the app exposes camera features to other apps, without mentioning it in the description. This guideline was applied to classify 17 vulnerable apps.

In some cases, multiple guidelines apply. For example, *com.bimandika.Congratulationsmalonepost* misuses a library without checking for permission and it does not validate the sender for system intents.

Based on these results, we formulate the subsequent answer to RQ$_1$:

> *Analyzing 1,258 apps, PREV reported 30 vulnerable apps without any false alarm (Precision=100%). The implication is that security analysts could use PREV to precisely identify vulnerabilities without any false alarm.*

8.4 RQ$_2$: Cost

We investigate the cost of *PREV* in terms of analysis time. First we discuss the cost of learning permission re-delegation models:

Before testing any given AUT, the permission re-delegation models were learnt by running our tool on 11,796 apps, as described in Section 5. *PREV* took 250 hours of CPU time to learn the models from 11,796 training apps. The bottleneck was static analysis. Most of the time was spent on extracting reachable APIs (approximately 1.275 minutes per app). Natural language processing, clustering, and model inference was quite fast (a magnitude of minutes in total). Since Android is always evolving (e.g., change of permission mechanism), the models should be updated whenever newer set (or versions) of reference apps become available. But, first of all, this training step does not have any real-time requirement. For testing a given set of apps under test, this step needs to be conducted only once. Model update can also be done incrementally, by analyzing a new app or a new version as soon as it is posted on the app store. It is not necessary to conduct static analysis on the whole training set. This would reduce the training time to significantly less than 250 hours. Moreover, in this experiment, we used a personal computer; however in actual industrial settings the cloud could be used instead and the analysis of distinct apps can be scheduled as independent jobs on distinct cloud hosts. Hence, we consider this training cost as affordable.

Next, we discuss the cost of analyzing a given AUT:

Outlier detection was performed on all the 1,258 apps; test case generation occurred only on the 77 apps that were reported as potentially vulnerable by the outlier detection phase of *PREV*. The time taken to perform outlier detection analysis is displayed in the boxplot in Fig. 15. As shown in the boxplot, on average, it took less than one minute and a half to complete the analysis. Only a few apps took a longer time (represented as outlier dots in the boxplot) and this was due to the use of complex libraries in the AUT.

Time taken to perform test case generation is shown in Fig. 16. Test case generation took longer than outlier detection; on average it takes less than 25 minutes per app. This is due to the genetic algorithm exploring the search space when trying to find an executable scenario that represents proof of concept attack. This duration depends on the timeout setting in the experimental configuration.

Considering these results we formulate the subsequent answer to RQ$_2$:

> *Outlier detection phase took 1.275 minutes on average. Test case generation phase took 25 minutes on average, but it is performed only for apps reported by outlier detection. It will take longer to use PREV when the models need to be updated. But this is typically an offline activity with no real-time requirement. Therefore, in general it would only take a magnitude of minutes to determine if an app is vulnerable. Overall we consider that the cost of using PREV is affordable in practice.*

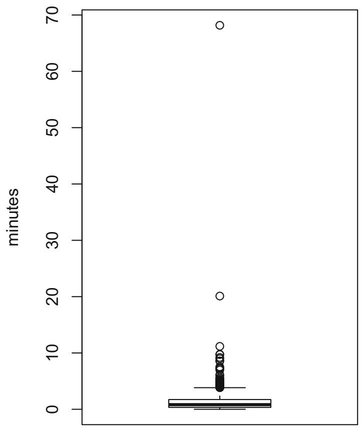


Fig. 15 Time (in minutes) taken for outlier analysis (above) and descriptive statistics (below)

mean	median	sd
1.28	0.82	2.34

8.5 RQ$_3$: Recall

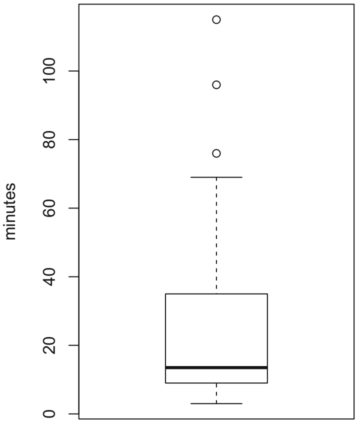
To answer RQ$_3$, we evaluate if *PREV* misses any vulnerabilities. We use two sets of *known vulnerable* apps. The first set of apps consists of 20 mutated apps which were subject to mutation. The second set of apps are 35 apps that were used earlier in Section 8.3.

Mutation tools are available to inject artificial faults in Java code, such as *Major* (Just 2014), *Pit* (Coles et al. 2016), *muJava* (Ma et al. 2005), *JavaLanche* (Fraser and Zeller 2011) and *Mdroid+* (Moran et al. 2018). They would represent valuable resources to construct an independent benchmark to investigate RQ$_3$. However, we found that these mutation tools are not compatible with our experimental settings. Most of them are not specific to Android and they produce programming errors when mutating arithmetic expressions, boolean conditions in decision points, types and references. *Mdroid+* (Moran et al. 2018) is the only Android specific mutation tool. It supports a set of mutation operators for producing realistic faults in Android apps. However, we found that those faults do not relate to permission re-delegation vulnerabilities.

Thus, we had to develop our own mutation tool. We defined two mutation operators for generating permission re-delegation vulnerabilities that reflect real-world vulnerabilities reported by Felt et al. (2011). We built a tool that applies these two mutation operators to inject permission re-delegation vulnerabilities in Android apps. Mutated apps are then subject to analysis using *PREV*. Since defining these operators and generating mutants is out of the scope here, we leave the details of this process in Appendix A.

Regarding the first set of apps, we started by a random sample of apps, consisting of 50 apps from the official store and 80 apps from open-source repository. We apply mutations to these apps to inject permission re-delegation vulnerabilities. Since an app might contain multiple components and multiple privileged APIs, the same operator might create several distinct mutated versions for the same app.

Fig. 16 Time (in minutes) taken for test case generation (above) and descriptive statistics (below)



mean	median	sd
24.9	13.5	25.3

However, due to implementation limitations, our mutation operators sometimes fail in generating a working vulnerability in the following cases:

- The mutated app crashes;
- The privileged API call is in a path that is not realizable from public entry points, due to certain path conditions in place;
- A path involves a UI event, such as a click event;
- A mutated component only accepts intents sent by the system.

We manually checked the mutants and discarded such cases.

Finally, our first set of apps consists of 20 mutants vulnerable by construction — 11 from closed source apps and 9 from open source apps — generated from 14 different original apps. Each mutant contains one injected vulnerability; some mutants are generated from the same original app but injected with different vulnerabilities.

The results of *PREV* on this benchmark is shown in Table 5. The top part shows the results corresponding to the mutants of closed source apps and the bottom part shows the results corresponding to the mutants from open source apps. The mutant name (first column) is a concatenation of the original app name, the mutation operator, and the unique-id of the component subject to mutation. As shown in Table 5, three distinct mutants were generated from the same app *com.nextcloud.client*. A tick-mark "✓" is present in the second column (TP) when *PREV* generated a test case for the corresponding mutant. Conversely an x-mark "✗" is reported in the third column (FN) when no test case was generated.

As shown in Table 5, among the mutated closed source apps, *PREV* detected 7 out of 11 vulnerable apps but missed four. Among the mutated open source apps, *PREV* detected 8 out of 9 vulnerable apps but missed one. In the following we discuss those missed cases.

PREV failed in generating a successful test case for `lwcr46lion.lwp_exposed_360` because the app expects a media URL (e.g., a URL pointing to .mp4 file) with an advertisement to display. When apps are expecting an intent *$data* field with a URI that

Table 5 Vulnerability report of *PREV* for mutated apps

Mutant	TP	FN
com.colortime.mandala_exposed_99	✓	
com.compasskeyboards.skullkeyboards_exposed_63	✓	
com.khampat.damdawi.in_exposed_63	✓	
com.khampat.damdawi.in_exposed_158		✗
com.khampat.damdawi.in_exposed_165	✓	
com.kisstakoala.vehiclesfree_direct_1	✓	
com.mademin.avoidthecircles_direct_1	✓	
com.mademin.avoidthecircles_exposed_47		✗
com.mfoundry.mb.android.mb_252070299_exposed_331	✓	
com.tdelphiblog.LazyShaker_exposed_33		✗
lwcr46lion.lwp_exposed_360		✗
com.futurice.android.reservator_17_exposed_2	✓	
com.junjunguo.pocketmaps_8_exposed_22	✓	
com.newsblur_138_direct_1	✓	
com.newsblur_138_direct_5	✓	
com.nextcloud.client_10040299_exposed_13		✗
com.nextcloud.client_10040299_exposed_81	✓	
com.nextcloud.client_10040299_exposed_108	✓	
net.mypapit.mobile.myposition_12_direct_1	✓	
org.ligi.gobandroid_hd_258_exposed_35	✓	

meets some conditions, these conditions are usually specified in the intent-filter. As discussed in Section 7.2, the test case generation phase relies on intent-filters in order to seed the *$data* field. However, this component does not specify an intent-filter at all (only the attribute `exported` is set to `true`). While the test case generation can seed other intent fields, such as `$action` and `$extra`, from the component's code even if they are not specified in the manifest file, the `$data` field is seeded either from the intent-filter or the component code *only* when the specification exists in the manifest file. As this component does not specify an intent-filter, our approach failed in generating the `$data` field that was essential to test this app. Similarly, the remaining 4 apps require inputs of specific data structures that could not be generated automatically and therefore, are missed.

Regarding the second set of vulnerable apps, we first looked at the 77 apps, used in Section 8.3, that were reported by our outlier detection phase. *PREV* correctly reported 30 apps as vulnerable (as discussed in Section 8.3). The remaining 47 apps were not reported as vulnerable because our final test generation phase was unable to generate proof-of-concept test cases. Out of these 47 apps, 16 are open source apps and thus, we were able to manually inspect the source code of these 16 apps. Manual investigation revealed the following:

- Five apps are actually vulnerable but missed by our tool. The reason is because our test generator was unable to generate the intent messages as required by those apps due to the similar problems explained above (missing intent-filter specifications);
- Four apps are not exploitable as the components are protected by custom permissions;

- Seven apps involve UI event-based paths that include user interactions (such as touches). Hence, they are not considered vulnerable (see Precondition PR_1 in Section 3.2).

In summary, the first set of apps contains 20 vulnerable apps. *PREV* detected 15 of them and missed five vulnerable apps. The second set of apps contains 35 vulnerable apps. *PREV* detected 30 of them and missed five vulnerable apps. Considering these results we formulate the subsequent answer to RQ_3:

PREV detected 45 out of 55 vulnerable apps (Recall=81.8%). The implication is that security analysts can use PREV to detect 81.8% of the apps containing permission re-delegation vulnerabilities.

8.6 RQ_4: Comparison

To investigate RQ_4, we compare *PREV* with *Covert* (Bagheri et al. 2015) and *IccTA* (Li et al. 2015).

We chose to compare our approach with *Covert* because it is designed to detect privilege escalation and permission re-delegation is a type of privilege escalation. It is a compositional analysis tool where a set of apps is analyzed together to see if there is a potential composite ICC vulnerability.

IccTA is not specifically designed to detect permission re-delegation vulnerabilities. It is, however, a widely-used generic tool built for various program analysis purposes for Android apps. It uses static taint analysis, a mainstream technique for various security analyses such as data leaks and privilege escalation. In principle, when configured with appropriate sources and sinks, it can be used to detect permission re-delegation vulnerabilities. *IccTA* implements static taint analysis approach that analyze inter-component communication (ICC). It is built on top of FlowDroid (Arzt et al. 2014) and IC3 (Octeau et al. 2015). IC3 is used to resolve targets in ICC, while FlowDroid is used to perform static taint analysis. In *IccTA*, we configured the ICC APIs (e.g., `getIntent()`) as sources and configured all the APIs that require special permission as sinks (listed in Pscout (Au et al. 2012)). If there is a data flow from a source (i.e., data sent from another app or another component) to a sink (i.e., performing privileged action), it is a case of permission re-delegation. We configured *IccTA* to report such cases. We note that such permission re-delegation cases are not necessarily vulnerabilities. Some of these cases could be the intended features of the app and thus, safe cases. ICC is a feature of Android framework. On the other hand, this in fact motivates the need of an approach like ours, for more precise vulnerability detection. In the following, we compare the results by discussing what cases are genuine vulnerabilities and what cases are safe cases.

We ran *Covert* and *IccTA* on the open source apps (595 open source apps that we used in Section 8.3 and the 20 mutated apps that we used in Section 8.5). We ran these tools on the closed source apps as well but we shall only discuss their results based on the analysis of open source apps. This is because these tools reported a large number of vulnerabilities in the closed source apps and it was difficult to verify them as we cannot inspect the source code and the tools do not generate proof-of-concept test cases. For each vulnerability report generated by these tools, we manually inspected the source code to establish the ground truth, i.e.,classify the report as a real vulnerability or as a safe case.

Table 6 Vulnerability report of *Covert* on open source apps

App	Vulnerable	Intentional Behaviour	Intra-comp. Intent	Private Components	System Intent
be.brunoparmentier.openbikesharing.app					✗
com.briankhuu.nfcmessageboard				✗	
com.duckduckgo.mobile.android		✗			
com.hectorone.multismssender			✗		
com.mschlauch.comfortreader		✗			
com.newsblur		✗			
com.seafile.seadroid2		✗			
com.xperia64.timidityae		✗			
de.hirtenstrasse.michael.lnkshortener		✗			
de.jkliemann.parkendd		✗			
de.syss.MifareClassicTool			✗		
de.yazo_games.mensaguthaben		✗			
net.kervala.comicsreader		✗			
org.glucosio.android		✗			
org.marcus905.wifi.ace			✗		
org.nerdcircus.android.klaxon		✗			
org.tigase.messenger.phone.pro	✓				
se.anyro.nfc_reader		✗			

Results of Covert The results of *Covert* on the open source apps is shown in Table 6. The first column shows the apps reported as vulnerable by *Covert*. The second column shows whether the reported vulnerable app is actually vulnerable. The remaining columns show whether the report is a safe case categorized as "Intentional Behaviour", " Intra-comp. Intent", "Private Components" or "System Intent". From our manual analyses, we observed that safe cases are cases of *app's intentional behaviour*, cases that can be activated only with an *intra-component intent*, cases that involve only *private components*, or cases that can be activated only with *system intent*.

Covert reported hundreds of vulnerabilities in the open source apps. However, only 18 of them are related to permission re-delegation. Out of these 18 vulnerabilities, only one is a real vulnerability and the rest are all safe cases. We manually verified and determined the safe cases as follows:

- Intentional behavior: some reported apps receive data from other apps (components) and use privileged APIs. These are cases of permission re-delegations. However, our inspection found that those cases actually implement app features declared in app's descriptions (intended features). For example, `com.newsblur`, `com.mschlauch.comfortreader`, `com.duckduckgo.mobile.android` and `se.anyro.nfc_reader` are browser, NFC reader and news/document reader apps, respectively; and browsing and data reading features are clearly declared in their Play Store descriptions. Therefore, those reports are actually safe cases of permission re-delegation. Most reports fall under this category.
- Intra-component intent: for some reported apps, the intent can only come from a component within the same app (i.e., result of `startActivityForResult` call).

Hence, those reports are actually safe cases, because the intent originates from the same app.
- Private components: for the reported app `com.briankhuu.nfcmessageboard`, the components in question are not exported. Therefore, they are only accessible within the same app and thus, not exploitable.
- System intent: for the reported app `be.brunoparmentier.openbikesharing.app`, the intent returned from the system component, `AccountManager`, is reported to be potentially dangerous. Since the intent actually comes from the Android system, we consider this as safe.

Regarding the 20 mutated apps, *Covert* did not report any of them as vulnerable. Thus, it missed all the vulnerabilities.

Results of IccTA Table 7 presents the results of running *IccTA* on our open source apps dataset. *IccTA* produced several reports for most apps. After manually investigating each report, we found that all the reports are not cases of permission re-delegation vulnerabilities. Those reports are cases of *app's intentional behaviour*, cases that involve only *private components*, cases that involve *user interaction*, or cases of *overtainting*. The first two types of cases are the same as *Covert*'s. In the following, we explain the other two types of cases.

- User interaction: some reported apps such as `com.newsblur` and `com.ringdroid` require the user to interact. If a user is involved, it is either an intended behavior or an action that can be aborted by the user. Therefore, we do not consider this as a vulnerability (See *Precondition* PR_1 in Section 3.2).
- Overtainting: for some reported apps such as `de.syss.MifareClassicTool`, the result of *IccTA* is affected by overtainting. For example, an activity instance containing an untrusted field is tainted. This instance is then used in a callback function but the

Table 7 Vulnerability report of IccTA on open source apps

App	Vulnerable	Intentional Behaviour	Private Components	User Interaction	Overtainting
com.alfray.timeriffic			✗		
com.commonsware.android.arXiv			✗		✗
com.newsblur				✗	
com.mschlauch.comfortreader		✗			
com.ringdroid				✗	
cz.romario.opensudoku		✗			
de.jkliemann.parkendd		✗			
de.syss.MifareClassicTool					✗
mobi.boilr.boilr			✗		
moe.minori.pgpclipper			✗		
org.jfet.batsHIIT				✗	
org.sixgun.ponyexpress			✗		
org.smc.inputmethod.indic				✗	
se.anyro.nfc_reader				✗	✗
sk.halmi.fbeditplus				✗	

field does not actually influence the invocation of any privileged API; hence this is a safe case.

As shown in Table 7, some reported cases correspond to more than one category of safe cases. Regarding the 20 mutated apps, *IccTA* did not report any of them as vulnerable. Thus, it missed all the vulnerabilities.

Table 8 shows the summary of results among *PREV*, *Covert*, and *IccTA*. Column *'Detected'* refers to the number of real permission re-delegation vulnerabilities that are detected; Column *'Safe'* refers to the number of cases that are reported as permission re-delegation vulnerabilities, but are considered safe according to our safe-cases rationale described above; Column *'Missed'* refers to the number of vulnerabilities in mutated apps that are missed.

Considering these results we formulate the subsequent answer to RQ$_4$:

> *PREV significantly outperforms Covert and IccTA in detecting permission re-delegation vulnerabilities because, according to our definition of permission re-delegation vulnerability, those tools missed the vulnerabilities detected by PREV, except the one vulnerability detected by Covert.*

8.7 RQ$_5$: Robustness

PREV detects vulnerable apps based on the permission re-delegation models that are learned on a large number of “safe” (benign and non-vulnerable) training apps. The detection accuracy might degrade when the quality of training apps degrades. Even though we carefully selected the “safe” apps (see Section 5), there is still a risk that some apps with security defects are included in the training set. In this case, our models may characterize these defects and *PREV* would not detect them as anomalies according to these models.

However, since *PREV* adopts a threshold-based algorithm, we made the assumption that it is robust against the inclusion of a *small* number of non-safe apps in the training set. Here we validate this assumption and quantify the robustness against the presence of non-safe apps in the training set.

Given vulnerability v, occurring in the app a that was assigned the cluster c, we define *robustness(v)* as the number of occurrences of vulnerability v that need to be included in the training set (in cluster c) to cause *PREV* not able to detect v.

To this aim, we gradually degrade the training set by replacing the safe apps with *contaminated apps* — apps that contain the same vulnerable permission re-delegation behaviors as those apps *PREV* correctly detected as vulnerable in Section 8.3. Learning is repeated

Table 8 Summary of comparison between *PREV*, *Covert*, and *IccTA*

	Open source apps		Mutated apps	
Tool	Detected	Safe	Detected	Missed
PREV	7	0	15	5
Covert	1	17	0	20
IccTA	0	15	0	20

on this new, degraded training set and the same experiment as in Section 8.3 is repeated, to assess if *PREV* is still able to detect the same vulnerabilities despite the contaminated training set. If *PREV* can still detect the same vulnerabilities, more and more contaminated apps are added to the training set and the process is iterated, until *PREV* can no longer detect the vulnerabilities; the number of contaminated apps at the last iteration gives us a robustness measure with respect to vulnerability v and cluster c.

More specifically, we consider the API frequency matrix M that was originally constructed at model inference step for a given cluster (see Section 5.3). An example of this matrix is shown in Fig. 17a. Rows represent apps in the training set and columns represent privileged APIs; a cell contains the value 1 when the API in the column is exposed (reachable from a public entry point) and the value 0 if the API is not exposed/used in the app.

Camera.open() in Fig. 17 is a privileged API associated with a permission re-delegation vulnerability that *PREV* detected on the subject app `com.appportunity.androidpreviewer`. *PREV* detected the vulnerability based on the permission re-delegation model learnt on the training apps of the cluster to which the subject app belongs (cluster 20). Now, we degrade this training set by modifying the API *Camera.open()* as reachable in *TrainingApp*$_2$, a training app from the same cluster as `com.appportunity.androidpreviewer`. This in fact corresponds to changing a value from 0 to 1 in the third column of the matrix. The change is highlighted in boldface in Fig. 17b. In this way, we contaminate the training set, by making the privileged but uncommon API *Camera.open()* more frequent and, thus, less likely anomalous.

In this experiment, *PREV* still detected the vulnerability due to the reachable API *Camera.open()* when the original training set is degraded with *one contaminated app*; but it was not able to detect the vulnerability anymore when it is degraded with *two contaminated apps*.

The experimental results are shown in Table 9. We can observe that our approach can still detect the same vulnerable apps (as using the original training set) even if the training set includes a few contaminated apps, ranging from 2-7 apps depending on the cluster. The table also shows the sizes of the clusters to which the vulnerable apps belongs.

These data are also shown as histogram in Fig. 18. It displays the distribution of robustness for the different vulnerabilities that *PREV* detected in official apps. For the majority of the vulnerabilities — 13 apps corresponding to the highest bar in Fig. 18 — *PREV* has robustness of five. The lowest robustness is two, which is observed for 4 apps. The highest robustness is seven, which is observed for 4 apps.

	openConnection()	connect()	**Camera.open()**	setWifiEnabled()
TrainingApp$_1$	1	1	1	0
TrainingApp$_2$	1	1	**0**	0
TrainingApp$_3$	1	1	0	0
TrainingApp$_4$	1	0	0	1

(a)

	openConnection()	connect()	**Camera.open()**	setWifiEnabled()
TrainingApp$_1$	1	1	1	0
TrainingApp$_2$	1	1	**1**	0
TrainingApp$_3$	1	1	0	0
TrainingApp$_4$	1	0	0	1

(b)

Fig. 17 API frequency matrix M. 1=the app exposes the API, 0=otherwise. **a** Original training set. **b** Contaminated training set

Table 9 Numbers of contaminated apps in training set that cause *PREV* fail to detect vulnerabilities

App	Number of Contaminated Apps	Cluster size
bestvalleygames.turningvalley	5	443
com.akgun.uknews	7	337
com.appportunity.androidpreviewer	2	438
com.appreka.mycoop	5	537
com.appsdv.smsmefitr	3	410
com.aurorasi.aurorasfa	5	537
com.bimandika.Congratulationsmalonepost	3	410
com.braingen.devanagarinotepad	5	469
com.dinosaur.dinosaur_vs_zombie	5	443
com.fmplural.radio	3	293
com.innogang.kollywoodNews	7	337
com.javirurro.games.spaceshipzigzag	3	410
com.josejoaquin.traductor	3	311
com.magmamobile.game.SpiderSolitaire2	3	410
com.netdania	5	537
com.npes87184.s2tdroid	7	337
com.rbsoftware.pfm.personalfinancemanager	5	537
com.reverbnation.artistapp.i739749	7	337
com.softdx.qrscanner	5	537
com.superfanu.bryantbulldogrewards	3	410
com.vent	5	537
lv.delfi.ru	5	469
piproduction.frankthejew	3	410
com.mendhak.gpslogger	4	380
com.seafile.seadroid2	2	537
org.ligi.ajsha	2	537
org.linphone	2	380
org.tigase.messenger.phone.pro	5	453
org.totschnig.myexpenses	5	537
org.ttrssreader	5	410

From Table 9, we can compute that the median value for the number of contaminated apps is 5 and the median cluster size is 424. It means that most of the vulnerable apps can still be detected if there are less than 1.18% of apps in each training cluster, which have the same vulnerabilities (based on the median values).

Considering these results we formulate the subsequent answer to RQ$_5$:

PREV is robust against the inclusion of non-safe apps in the training set to a certain extent. To bypass PREV, there must be a few apps (1.18% of apps in the cluster) that have the same vulnerability as the AUT (i.e., the same re-delegated API is used) and that fall into the same cluster as the AUT.

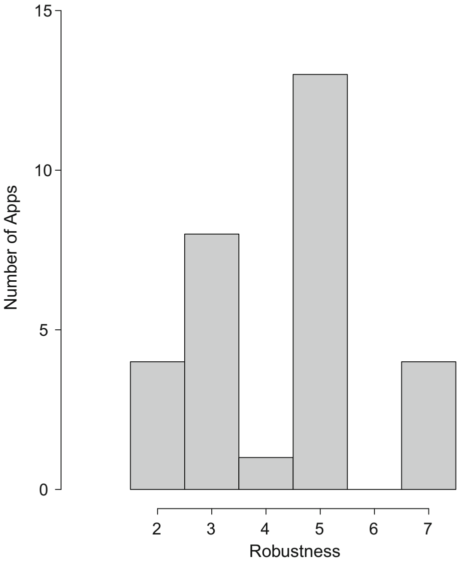


Fig. 18 Histogram showing the levels of training set contamination

8.8 RQ$_6$: Threshold

The vulnerability detection algorithm is based on the boxplot approach, and a case is classified as an outlier when its distance d is larger than the threshold $t_{outlier}$. We are interested in studying the impact of different thresholds on the vulnerability detection capability of our approach.

To investigate this research question, we consider the list of apps correctly detected as vulnerable in Section 8.3. It consists of 30 apps (either open source and closed source), which have been classified as vulnerable by *PREV* and have been manually verified.

The experiment consists of changing the threshold value and verifying how many of these apps can still be detected as vulnerable. We expect that the larger the threshold we use, the fewer vulnerable apps would be detected.

To compute new threshold values, we add a multiplication factor α in the definition of threshold from Section 5.3. The threshold definition is updated as follows:

$$t_{outlier} = Q_3 + \alpha 1.5(Q_3 - Q_1)$$

If $\alpha = 1$, we have our original definition of threshold. New threshold values are obtained by using different values of α, and are used to repeat the classification procedure.

Figure 19 shows the results of this experiment. It shows the true positives against the increasing values of α. The number of true positives is the same until $\alpha \leq 1.33$. It then drops significantly and reaches 17 (nearly half of the initial value of 30) at $\alpha = 1.6$. Eventually, no vulnerability can be detected at $\alpha \geq 2$.

Considering these results, we can answer to RQ$_6$ in this way:

PREV is sensitive to the threshold used to detect outliers. In particular, the number of detected vulnerabilities drops to almost half when increasing the threshold by a 30% factor, and no app is classified as vulnerable with a threshold increased by a 100% factor.

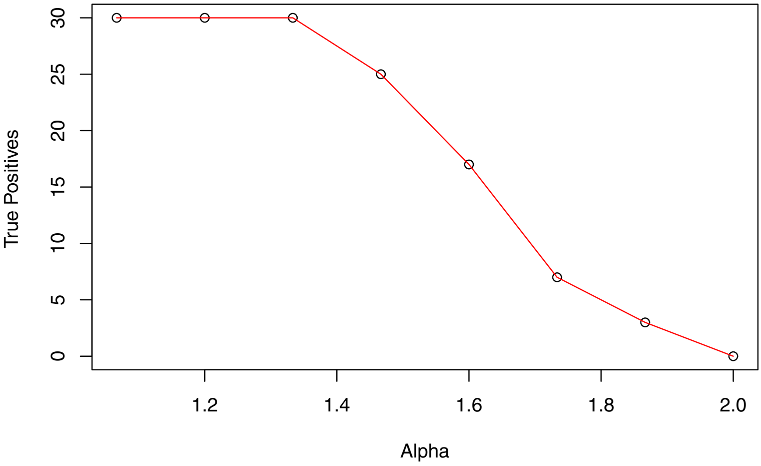


Fig. 19 Impact of different thresholds on vulnerability detection

8.9 Limitation and Discussion

Despite the positive results obtained, our current approach may be affected by some limitations that we discuss here:

Learning from Training Apps If the training apps we use are not sufficient or representative of real world, safe apps, the inferred models would be incomplete. Consequently, our approach may be imprecise. Incomplete training set could miss some cases of legitimate permission re-delegation. Training apps in the same clusters having the same vulnerable permission re-delegation behaviors would limit the capability of our tool to detect vulnerabilities. This could be the case when developers copy-paste the same piece of vulnerable code from unreliable sources or when they include vulnerable libraries in their apps.

Our counter argument is that, as we motivated in Section 1, apps re-delegating and executing privileged tasks is generally a feature itself. It could be a false alarm if an app is flagged as vulnerable just because the app performs such an action. As such, learning legitimate permission re-delegation behaviors from reference apps is one major design of our approach to avoid many false alarms, despite the trade-off of false negatives. Our current best effort to mitigate this trade-off is using only apps from top, most-downloaded apps from well-reputed companies as the reference apps; since the learning requires tens of thousands of apps, it is impractical to manually inspect all of them and make sure they are all actually "safe". In our future work, we plan to mitigate the problem of apps with vulnerable libraries by incorporating recent vulnerable library detection approaches (Backes et al. 2016; Ma et al. 2016) and by excluding them from training.

App Descriptions Our approach is based on the apps descriptions available in the App Store for learning the permission re-delegation models. An app description typically describes about the features offered by the app. Our approach relies on this information to group apps with similar features. However we do not assume that app description describes much about permission re-delegation behaviors, because they may be implicit. Our assumption here is that, for training apps, their apps descriptions are consistent with their permission re-delegation behaviors, i.e., reference apps with similar features implement similar permission re-delegation behaviors. It could result in false positives and false negatives when a group of training apps with a similar description implements different permission re-delegation behaviors.

Evasion Our approach requires that the app under test contains a description that is (i) in English and (ii) with a minimum length of 10 words. When its description is not informative or not detailed enough to assign it to the correct cluster, a sub-optimal cluster might be identified for the app, leading to incorrect classification of its behavior. Hence, a malicious app may evade our analysis by negating either of these requirements. To satisfy the first requirement, automatic translation tools could be used to obtain corresponding English text for a given app description in a different language. When an app evades the second requirement, e.g., by publishing the app with no description, it would be suspicious and is unlikely to gain trust from end-users. Furthermore, when an app store is interested in using our approach, the app store could enforce these two requirements and reject apps that do not meet them. Honest app developers may also be willing to scan her/his app for vulnerabilities before publishing the app and thus, they would have an incentive to provide appropriate app descriptions.

Static Analysis Like many other static analysis-based approaches, our static analysis suffers from its inherent weaknesses. Call graphs generated by our underlying static analysis tool (FlowDroid) may not be sound and complete. The tool may miss call edges in the presence of complex code such as obfuscated code, native code and reflection and dynamic code loading. This could result in false negatives when the missing edges are those from public entry points to a privileged API. It may also generate spurious additional edges for code such as Thread runnables. In this case, it may result in false positives.

Our approach does not perform taint analysis or consider data sanitization functions that sanitize input data from other apps or discard them when the data violates security policies. It only analyzes control dependencies and reachability in the call graph, according to the threat model we consider (Section 3). Our approach reports a privileged API call as a vulnerability even if it may not use the data controlled by the attacker. This is intentional and designed to avoid false negatives, because permission re-delegation vulnerabilities do not necessarily result from data flows and data usage. A false positive could happen when the privileged action is performed using the input data only after they have been sanitized. However, our empirical result of zero false positive suggests that our anomaly detection phase helps mitigate such false positives, by learning those safe cases from the training apps.

Test Generation Like any other test generation-based approach, the code coverage of our test generator is also limited. Although we apply genetic algorithm, a state-of-the-art technique for test generation, it may still not be able to generate proof-of-concepts for some of the target paths, possibly resulting in false negatives. Our test generator can be improved if seed values can be more accurately identified. This can be done by modeling string operations and solving constraints on string values. In future work, we plan to combine our genetic algorithm-based test generator with a string constraint solving technique such as Thome et al. (2020) for more effective test generation.

Supported Components Our approach analyzes Activities, Broadcast Receivers and Services. But it does not analyze Content Providers. Content Providers manage access to app data and are more subject to data leak issues. However, our approach focuses on permission re-delegation issues that leak capabilities/privileges. Detecting data leaks from Content

Providers is out of our scope. Due to the incompleteness problem of the underlying static analysis tool highlighted above, our current implementation cannot handle dynamically registered Broadcast receivers as well. But this is rather a limitation of the tool not the approach.

Dynamic Permissions In recent versions of Android, even if permissions are still required to be declared in the manifest, they can be granted and revoked by the user during runtime. Our approach does not model this dynamic revocation possibility. Instead, our approach assumes the most dangerous scenario of an end-user who is not security aware, where all the permissions requested are always granted and, thus, if they are exposed, a permission re-delegation vulnerability is reported.

To summarize, our approach is neither sound nor complete due to various practical limitations of program analysis and machine learning. However, based on our empirical results, we can argue that our tool resulting from such an approach can automatically detect many permission re-delegation vulnerabilities with a very low false alarm rate (zero false alarm in our experiments) in a matter of minutes. Hence, the implication is that while our approach is neither sound nor complete, it has practical benefits since the detected vulnerabilities come at almost zero cost. As we make our tool and dataset publicly available, researchers can replicate and/or repeat the experiments to validate or refute our claims.

9 Related Work

Our work is related to the work that deals with permission re-delegation problems or information leaks (closely-related vulnerabilities) on smart phones.

Natural Language Processing and Machine Learning The approaches proposed in Mudflow (Avdiienko et al. 2015), Chabada (Gorla et al. 2014), and Anflo (Demissie et al. 2018) are closely related to ours. Mudflow (Avdiienko et al. 2015) uses static data flow analysis to learn data-flow patterns of apps. The main difference with our approach is that Mudflow learns data-flow patterns from all available benign apps and compares the patterns of the given app under test against all those learnt patterns, whereas our approach compares them only based on similar apps. We apply similar techniques as Chabada (Gorla et al. 2014) in terms of natural language processing of apps descriptions and clustering of apps. But the goals differ. The goal of their approach is to find anomalous apps among the apps in the wild. Ours specifically targets at detecting permission re-delegation vulnerabilities in a given app. Chabada reports anomalies based on the presence of privileged API calls only, whereas, we consider execution paths from public entry points to privileged API calls related to permission re-delegation. Anflo (Demissie et al. 2018) relies on apps descriptions to group similar apps together. However, Anflo uses only the dominant topic to group together similar apps whereas *PREV* considers all topics probabilities to improve clustering. The major difference with Anflo and *PREV* is that Anflo uses the information-flow model to find anomalous information flows, whereas *PREV* uses permission re-delegation model to find anomalous permission re-delegations. All the above-mentioned approaches do not incorporate dynamic analysis and test case generation, which is used by our approach to reduce false positives.

Permission Re-Delegation Another closely related work is the work by Felt et al. (2011), which first introduced the permission re-delegation problem. Chin et al. (2011), Lu et al.

(2012), and Zhong et al. (2012) also presented similar approaches. These approaches identify all possible entry points of an app and then perform data-flow analysis starting from an entry point until a privileged API is reached. However, Felt et al. and Chin et al. acknowledged that such approaches detect permission re-delegation cases but cannot distinguish between intended cases and vulnerable ones, and thus, possibly produce many false alarms. In our previous work (Avancini and Ceccato 2013) we also proposed two preliminary solutions for detecting permission re-delegation using either static or dynamic analysis. However, this work was incomplete because it could not distinguish between vulnerabilities and safe cases of permission re-delegation. Our new approach presents an extension of this threat model by imposing an additional precondition. It concludes the existence of permission re-delegation vulnerability in an app only when permission re-delegation in the app is inconsistent with permission re-delegation observed among similar apps. Empirical evidence suggests that our approach was accurate in detecting permission re-delegation vulnerabilities without producing false alarms.

Static Analysis Several static taint analysis-based approaches (Wei et al. 2014; Gordon et al. 2015; Sbîrlea et al. 2013; Octeau et al. 2013; Octeau et al. 2015; Tsutano et al. 2017; Mann and Starostin 2012; Li et al. 2015; Klieber et al. 2014; Au et al. 2012; Junaid et al. 2016; Bagheri et al. 2015; Xu et al. 2017; Bosu et al. 2017; Lu et al. 2015) have been proposed to detect information or privacy leaks in mobile apps. These approaches are related since they can be adapted to address the permission re-delegation problem. Tainted sources are system calls that access private data (e.g., global position, contacts entries), while sinks are all the possible ways that make data leave the system (e.g., network transmissions). An issue is detected when sensitive information from tainted sources could potentially leave the app through one of the sinks. In the following, we discuss some of the recent approaches and explain the major differences.

Droidsafe (Gordon et al. 2015) is a static information flow analysis tool that can detect potential information leaks in Android apps. Amandroid (Wei et al. 2014) detects privacy data leaks due to inter-component communication (ICC) in Android apps. Epicc (Octeau et al. 2013) identifies ICC connection points by using inter-procedural data-flow and points-to analysis. IC3 (Octeau et al. 2015) improves Epicc by using complex techniques to resolve targets and values used in ICC. Similar to Epicc, Tsutano et al. (2017) also analyzes interacting apps. However, instead of combining apps for analysis, it uses a static class loader that enables analysis of a large number of interacting apps. IccTA (Li et al. 2015) attempts to improve static taint analysis of Android apps in ICC by modeling the life-cycle and callback methods by instrumenting the code of the app. DidFail (Klieber et al. 2014) detects data leaks between activities through implicit intents. But it does not consider other components and explicit intents. Grace et al. (2012) perform static analysis in stock Android apps released by different vendors, to check the presence of any information leak. Since vendors modify or introduce their own apps, they might also introduce new vulnerabilities. The work, however, is limited to stock apps on specific vendor devices. Dexteroid (Junaid et al. 2016) reverse engineers life cycle models of Android components and detect privacy data leaks. AAPL (Lu et al. 2015) performs multiple specialized static analyses such as conditional flow identification and joint flow tracking for more accurate detection of privacy data leaks. It additionally employs a technique called peer voting to filter out legitimate privacy leaks. Peers are determined based on Google's app recommendation system.

All the above-mentioned approaches apply static analysis techniques on mobile code similarly to ours so as to detect information sources and sinks. Apart from the difference in the threat model, the major difference of our approach from the above-mentioned techniques is providing execution scenarios. We are particularly interested in accurately detecting permission re-delegation vulnerabilities. Therefore, for accuracy, we propose an approach that seamlessly incorporates machine learning and test case generation into static analysis.

Dynamic Analysis and Runtime Monitoring TaintDroid (Enck et al. 2010) is a tool for performing dynamic taint analysis. It relies on a modified Android installation that tracks tainted data at run-time. The implementation showed minimal size and computational overhead, and was effective in analyzing many real Android apps. ARF (Gorski and Enck 2019) detects permission re-delegation vulnerabilities in system services of the Android framework. Their approach detects when a system service that requires no or low permission to be called, acts as deputy and calls a second system service with high privilege. Instead of analyzing the Android framework, our approach applies to apps that are installed and run in user space. Zhang and Yin (2014) proposed Appsealer, a runtime patch to mitigate permission re-delegation problem. It performs static data-flow analysis to determine sensitive data-flows from sources to sinks and apply the patch before the invocation of a privileged API such that the app alerts the user of a potential permission re-delegation attack and requests the user's authorization to continue. This is an alternative way of distinguishing legitimate and anomalous permission re-delegation by relying on the user. Lee et al. (2017) proposed Sealant, which is similar to Appsealer; it additionally extends the Android framework to monitor vulnerable inter-app communication at run-time and prevents attacks. Roppdroid (Dai et al. 2017) mitigates permission re-delegation problems via resource virtualization and modified ICC mechanism. Bugiel et al. (2012) proposed a system-centric (rather than application-dependent) solution that conducts policy-driven runtime monitoring of communications between apps at middleware IPC layer and at kernel layer. This requires modification of Android framework. In contrast to these approaches, our approach applies machine learning across similar apps to infer anomalous permission re-delegation automatically. It does not rely on the end user, does not modify Android framework, and does not involve runtime monitoring.

Colluding Apps Covert (Bagheri et al. 2015) uses static analysis and formal verification to verify the collusive data leaks and privilege escalation problems that arise from the interaction of multiple apps (known as colluding apps). It infers the security properties from individual apps and reasons about the security of the phone device as a whole; that is, it checks whether apps installed in a device can collude with each other to generate attacks. The major difference between Covert and *PREV* is that Covert's formal verification process relies on predefined security policies that describe what is normal and what is abnormal. Similar to Covert, AppHolmes (Xu et al. 2017) also uses static analysis and predefined security policies to detect collusive data leaks. DIALDroid (Bosu et al. 2017) addressed the complexity problem of performing pairwise program analysis of apps to detect collusive data leaks by incorporating optimized, efficient data storage and fast query processing techniques into static analysis of ICC data-flows. The major difference with these approaches is that, instead of relying on predefined security policies, we compare the security properties of the app against functionally similar apps to detect abnormal/malicious behavior.

Test Case Generation FuzzDroid (Rasthofer et al. 2017) generates executable test cases using an evolutionary algorithm, with the aim of covering a given target location (e.g. privileged API call) to expose malicious behaviors of a given app. EvoSuite (Fraser and Arcuri 2011) is a general purpose test generator based on genetic algorithm, which can also be applied to generate executable scenarios for Android apps. App graphical user interface

(GUI) testing approaches (Hu and Neamtiu 2011; Amalfitano et al. 2012; Amalfitano et al. 2011; Mahmood et al. 2012) have been proposed to detect events and event sequences that make the apps crash and identify abnormalities of apps. EvoDroid (Mahmood et al. 2014) and Sapienz (Mao et al. 2016) apply search-based testing techniques to systematically test Android apps. EvoDroid uses a model generated based on static data collected from manifest file and layout XMLs to guide the search process. Similar to these approaches, we also apply a search-based algorithm to systematically explore app behaviors. The main difference is that they generate test cases mainly to detect abnormalities in apps such as crashes, exceptions, and violations of access permissions; our approach, instead, generates security test cases that expose permission re-delegation vulnerabilities in Android apps.

10 Conclusion

Smart phone apps are often developed under a high time-to-market pressure. As a result, they are often delivered with defects or vulnerabilities that may threaten the security and privacy of end users. In particular, apps that are granted with special permissions could expose privileged services to unprivileged apps, which may then exploit this to perform malicious actions without user knowledge. Automated support to detect such problems would be beneficial to security analysts and app developers.

In this paper, we present a novel approach to accurately detect and test permission re-delegation vulnerabilities by combining static analysis, natural language processing, machine learning, and genetic algorithm-based test generation techniques. Our approach detects vulnerable apps that abnormally expose privileged actions to other (potentially malicious) apps and distinguishes them from legitimate permission re-delegation cases. It also generates proof-of-concept attacks to prove the vulnerabilities and security reports to document them. We evaluated our approach on 1,258 real world apps from the official Google Play store. Our approach automatically detected 30 apps that are vulnerable to permission re-delegation attacks without any false alarm, significantly outperforming recent approaches — *Covert* and *IccTA* — that can detect permission re-delegation problems. The analysis was done in a matter of minutes.

In future work, we plan to improve both our static analysis and test generation phases, e.g., by handling string operations, for more effective vulnerability detection.

Acknowledgements The content of this paper is part of the PhD thesis of the first author Biniam Fisseha Demissie. The work of Lwin Khin Shar was supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

Funding Open access funding provided by Universit degli Studi di Verona within the CRUI-CARE Agreement.

Appendix A: Mutation Operators

In the following we explain the two mutation operators that we used in our controlled experiment (Section 8.5).

A.1 Mutation Operator μ_1: Expose API

To provide its services, an app may be granted with special permissions to perform privileged operations (APIs). If these operations are security-sensitive, the app developer may limit these operations among its internal components only and may not intend to accept action requests of these operations from external apps.

Mutation μ_1 is used to inject a security defect by contradicting the developer's intention of limiting the exposure of a privileged API to action requests. The precondition to apply this mutation operator is:

- There is an exposed component that can be called by other apps to request an action from this app;
- the app contains a call to a privileged API;
- but this privileged API is not exposed to action requests (i.e., the call to this API is not reachable from public entry points).

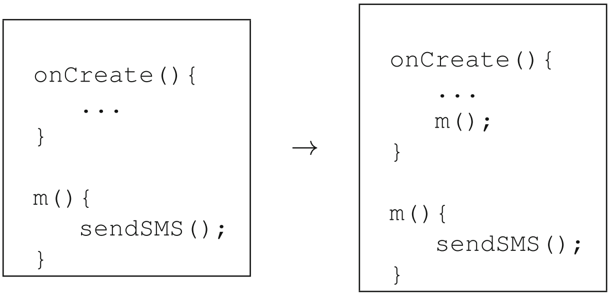
If this precondition is met by a given app, we apply μ_1 by exposing the privileged API to action requests via the exposed component. Hence, the postcondition after applying μ_1 is:

- The privileged API is now exposed (i.e., reachable from at least one public entry point) and thus, the app is vulnerable to permission re-delegation attacks.

For example, in Fig. 20, the left-hand side shows the original code of an app; *onCreate* is an exposed component; the method *m* contains a call to *sendSMS*, but *m* is not reachable from any public entry point. The right-hand side shows the mutated code, which exposes a privileged API by adding a call to the method *m* in the exposed component.

A.2 Mutation Operator μ_2: Expose Component

Similarly to μ_1, μ_2 is used to inject a security defect by contradicting the developer intention of limiting the exposure of a privileged operation. However, in this case, the mutation is on the component visibility to other apps.

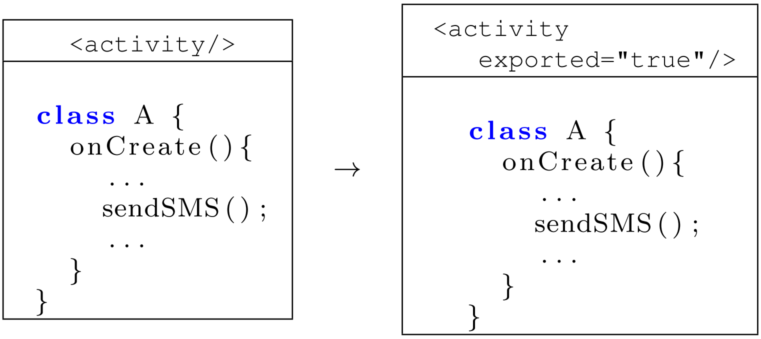


```
onCreate(){
    ...
}

m(){
    sendSMS();
}
```

→

```
onCreate(){
    ...
    m();
}

m(){
    sendSMS();
}
```

Fig. 20 Applying mutation operator μ_1: *Expose API*



```
<activity/>

class A {
  onCreate(){
    ...
    sendSMS();
    ...
  }
}
```

→

```
<activity
  exported="true"/>

class A {
  onCreate(){
    ...
    sendSMS();
    ...
  }
}
```

Fig. 21 Mutation operator μ_2: *Expose component*

The precondition to apply this operator is:

- A component is not exposed; so it cannot be called by other apps;
- the component contains a call to a privileged API;
- but the call to this API is not reachable from public entry points.

If this precondition is met by a given app, we apply μ_2 by changing the visibility of the component in the *manifest* file of the app. Hence, the postcondition after applying μ_2 is:

- The component is now exposed; so the privileged API call is now reachable from at least one public entry point.

For example, in Fig. 21, the left-hand side shows the original code; the method *onCreate* in the activity *A* is not exposed and it contains a call to a privileged API *sendSMS*, which is also not reachable from any public entry point. The right-hand side shows the mutated code, which specifies in the manifest file that activity *A* is publicly callable.

References

Al-Subaihin AA, Sarro F, Black S, Capra L, Harman M, Jia Y, Zhang Y (2016) Clustering mobile apps based on mined textual features. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, ESEM '16. ACM, New York, pp 38:1–38:10. https://doi.org/10.1145/2961111.2962600

Amalfitano D, Fasolino A, Tramontana P (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, pp 252–261, https://doi.org/10.1109/ICSTW.2011.77

Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM (2012) Using GUI ripping for automated testing of Android applications. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering, ASE 2012. ACM, New York, pp 258–261. https://doi.org/10.1145/2351676.2351717

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (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, PLDI '14. ACM, New York, pp 259–269. https://doi.org/10.1145/2594291.2594299

Au KWY, Zhou Y, Huang Z, Lie D (2012) Pscout: analyzing the Android permission specification. In: Proceedings of the 2012 ACM conference on computer and communications security. ACM, pp 217–228

Avancini A, Ceccato M (2013) Security testing of the communication among Android applications. In: Proceedings of the 8th international workshop on automation of software test. IEEE Press, pp 57–63

Avdiienko V, Kuznetsov K, Gorla A, Zeller A, Arzt S, Rasthofer S, Bodden E (2015) Mining apps for abnormal usage of sensitive data. In: Proceedings of the 37th international conference on software engineering. IEEE Press, pp 426–436

Backes M, Bugiel S, Derr E (2016) Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on computer and communications security, CCS '16. ACM, New York, pp 356–367. https://doi.org/10.1145/2976749.2978333

Bagheri H, Sadeghi A, Garcia J, Malek S (2015) Covert: Compositional analysis of android inter-app permission leakage. IEEE Trans Softw Eng 9:866–886

Blei DM, Ng AY, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res 3:993–1022

Bosu A, Liu F, Yao DD, Wang G (2017) Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: Proceedings of the 2017 ACM on Asia conference on computer and communications security. ACM, pp 71–85

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A, Shastry B (2012) Towards taming privilege-escalation attacks on android. In: NDSS, vol 17. Citeseer, p 19

Chin E, Felt AP, Greenwood K, Wagner D (2011) Analyzing inter-application communication in Android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services, MobiSys '11. ACM, New York, pp 239–252. https://doi.org/10.1145/1999995.2000018

Coles H, Laurent T, Henard C, Papadakis M, Ventresque A (2016) Pit: a practical mutation testing tool for java. In: Proceedings of the 25th international symposium on software testing and analysis, pp 449–452

Dai T, Li X, Hassanshahi B, Yap RH, Liang Z (2017) Roppdroid: Robust permission re-delegation prevention in android inter-component communication. Comput Secur 68:98–111

Demissie BF, Ghio D, Ceccato M, Avancini A (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, pp 255–266

Demissie BF, Ceccato M, Shar LK (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

Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the em algorithm. J R Stat Soc Series B Methodol 1–38

Enck W, Gilbert P, gon Chun B, Cox LP, Jung J, McDaniel P, Sheth AN (2010) Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: 9Th usenix symposium on operating systems design and implementation

Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of Android application security. In: Proceedings of the 20th USENIX conference on security, SEC'11. USENIX Association, Berkeley, pp 21–21. http://dl.acm.org/citation.cfm?id=2028067.2028088

Felt AP, Wang H, Moshchuk A, Hanna S, Chin E (2011) Permission re-delegation: attacks and defenses. In: 20Th usenix security symposium

Fraser G, Arcuri A (2011) Evosuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering, ESEC/FSE '1. ACM, pp 416–419

Fraser G, Zeller A (2011) Mutation-driven generation of unit tests and oracles. IEEE Trans Softw Eng 38(2):278–292

Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC (2015) Information flow analysis of android applications in droidsafe. In: NDSS, vol 15, p 110

Gorla A, Tavecchia I, Gross F, Zeller A (2014) Checking app behavior against app descriptions. In: Proceedings of the 36th international conference on software engineering. ACM, pp 1025–1035

Gorski S. A. III, Enck W (2019) Arf: identifying re-delegation vulnerabilities in android system services. In: Proceedings of the 12th conference on security and privacy in wireless and mobile networks, pp 151–161

Grace MC, Zhou Y, Wang Z, Jiang X (2012) Systematic detection of capability leaks in stock Android smartphones. In: NDSS. The Internet Society. http://dblp.uni-trier.de/db/conf/ndss/ndss2012.html

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD Explor Newslett 11(1):10–18

Hodge VJ, Austin J (2004) A survey of outlier detection methodologies. Artif Intell Rev 22(2):85–126

Holland JH (1975) Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence. University of Michigan Press, Ann Arbor

Hu C, Neamtiu I (2011) Automating GUI testing for Android applications. In: Proceedings of the 6th international workshop on automation of software test, AST '11. ACM, New York, pp 77–83. https://doi.org/10.1145/1982595.1982612

Junaid M, Liu D, Kung D (2016) Dexteroid:, Detecting malicious behaviors in android apps using reverse-engineered life cycle models. Comput Secur 59:92–117

Just R (2014) The Major mutation framework: efficient and scalable mutation analysis for Java. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, pp 433–436

Klieber W, Flynn L, Bhosale A, Jia L, Bauer L (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, SOAP '14. ACM, New York, pp 1–6. https://doi.org/10.1145/2614628.2614633

Laurikkala J, Juhola M, Kentala E, Lavrac N, Miksch S, Kavsek B (2000) Informal identification of outliers in medical data. In: Fifth international workshop on intelligent data analysis in medicine and pharmacology, vol 1, pp 20–24

Lee YK, Bang JY, Safi G, Shahbazian A, Zhao Y, Medvidovic N (2017) A sealant for inter-app security holes in android. In: Proceedings of the 39th international conference on software engineering, ICSE '17. IEEE Press, Piscataway, pp 312–323. https://doi.org/10.1109/ICSE.2017.36

Li L, Bartel A, Bissyandé T. F., Klein J, Le Traon Y, Arzt S, Rasthofer S, Bodden E, Octeau D, Mcdaniel P (2015) IccTA: Detecting inter-component privacy leaks in Android apps. In: Proceedings of the 37th international conference on software engineering (ICSE 2015), pp 280–291

Lu L, Li Z, Wu Z, Lee W, Jiang G (2012) Chex: Statically vetting Android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM conference on computer and communications security, CCS '12. ACM, New York, pp 229–240. https://doi.org/10.1145/2382196.2382223

Lu K, Li Z, Kemerlis VP, Wu Z, Lu L, Zheng C, Qian Z, Lee W, Jiang G (2015) Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In: NDSS

Ma Y, Offutt J, Kwon YR (2005) Mujava: an automated class mutation system. Softw Test Verif Reliab 15(2):97–133

Ma Z, Wang H, Guo Y, Chen X (2016) Libradar: Fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th international conference on software engineering companion, ICSE '16. ACM, New York, pp 653–656. https://doi.org/10.1145/2889160.2889178

Mahmood R, Esfahani N, Kacem T, Mirzaei N, Malek S, Stavrou A (2012) A whitebox approach for automated security testing of Android applications on the cloud. In: Proceedings of the 7th international workshop on Automation of Software Test (AST), pp 22–28

Mahmood R, Mirzaei N, Malek S (2014) Evodroid: Segmented evolutionary testing of android apps. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 599–609

Mann C, Starostin A (2012) A framework for static detection of privacy leaks in Android applications. In: 27Th Symposium on Applied Computing (SAC): computer security track, pp 1457–1462

Mao K, Harman M, Jia Y (2016) Sapienz: multi-objective automated testing for android applications. In: Proceedings of the 25th international symposium on software testing and analysis. ACM, pp 94–105

McCallum AK (2002) Mallet: A machine learning for language toolkit

Moran K, Tufano M, Bernal-Cárdenas C., Linares-Vásquez M., Bavota G, Vendome C, Di Penta M, Poshyvanyk D (2018) Mdroid+: a mutation testing framework for android. In: 2018 IEEE/ACM 40Th international conference on software engineering: companion (ICSE-companion). IEEE, pp 33–36

Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Le Traon Y (2013) Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In: Proceedings of the 22Nd USENIX conference on security, SEC'13. USENIX Association, Berkeley, pp 543–558. http://dl.acm.org/citation.cfm?id=2534766.2534813

Octeau D, Luchaup D, Dering M, Jha S, McDaniel P (2015) Composite constant propagation: application to android inter-component communication analysis. In: Proceedings of the 37th International Conference on Software Engineering (ICSE). http://siis.cse.psu.edu/pubs/octeau-icse15.pdf

OWASP (2015) OWASP mobile security project top 10. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-2015_Scratchpad

Porter MF (1997) Readings in information retrieval/ An algorithm for suffix stripping. Morgan Kaufmann Publishers Inc., San Francisco, pp 313–316. http://dl.acm.org/citation.cfm?id=275537.275705

Rasthofer S, Arzt S, Triller S, Pradel M (2017) Making malory behave maliciously: targeted fuzzing of android execution environments. In: Proceedings of the 39th international conference on software engineering, ICSE '17. IEEE Press, pp 300–311

Reps T, Horwitz S, Sagiv M (1995) Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '95. ACM, pp 49–61

Sadeghi A, Esfahani N, Malek S (2014) Mining the categorized software repositories to improve the analysis of security vulnerabilities. In: International conference on fundamental approaches to software engineering. Springer, pp 155–169

Sbîrlea D, Burke MG, Guarnieri S, Pistoia M, Sarkar V (2013) Automatic detection of inter-application permission leaks in android applications. IBM J Res Dev 57(6):10:1–10:12. https://doi.org/10.1147/JRD.2013.2284403

Thome J, Shar LK, Bianculli D, Briand L (2020) An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE Trans Softw Eng 46(2):163–195

Tsutano Y, Bachala S, Srisa-an W, Rothermel G, Dinh J (2017) An efficient, robust, and scalable approach for analyzing interacting android apps. In: Proceedings of the 39th international conference on software engineering, ICSE '17. IEEE Press, Piscataway, pp 324–334. https://doi.org/10.1109/ICSE.2017.37

Wegener J, Baresel A, Sthamer H (2001) Evolutionary test environment for automatic structural testing, vol 43, pp 841–854. https://doi.org/10.1016/S0950-5849(01)00190-2. http://www.sciencedirect.com/science/article/pii/S0950584901001902

Wei F, Roy S, Ou X, 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, CCS '14. ACM, New York, pp 1329–1341. https://doi.org/10.1145/2660267.2660357

Witten IH, Frank E, Hall MA (2011) Data mining: practical machine learning tools and techniques. Morgan Kaufmann, San Mateo

Xu M, Ma Y, Liu X, Lin FX, Liu Y (2017) Appholmes: detecting and characterizing app collusion among third-party android markets. In: Proceedings of the 26th international conference on World Wide Web, pp 143–152

Zhang M, Yin H (2014) Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications

Zhong J, Huang J, Liang B (2012) Android permission re-delegation detection and test case generation. In: 2012 International conference on computer science and service system, pp 871–874

Affiliations

Biniam Fisseha Demissie[1] · Mariano Ceccato[2] · Lwin Khin Shar[3]

Biniam Fisseha Demissie
demissie@fbk.eu

Lwin Khin Shar
lkshar@smu.edu.sg

[1] Fondazione Bruno Kessler, Trento, Italy

[2] University of Verona, Verona, Italy

[3] School of Information Systems, Singapore Management University, Singapore, Singapore