5-2018

AnFlo: Detecting anomalous sensitive information flows in Android apps

Biniam Fisseha DEMISSIE

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

Lwin Khin SHAR

AnFlo: Detecting Anomalous Sensitive Information Flows in Android Apps

Biniam Fisseha Demissie, Mariano Ceccato
Fondazione Bruno Kessler
Trento, Italy
{demissie,ceccato}@fbk.eu

Lwin Khin Shar
School of Computer Science and Engineering
Nanyang Technological University, Singapore
lkshar@ntu.edu.sg

ABSTRACT

Smartphone apps usually have access to sensitive user data such as contacts, geo-location, and account credentials and they might share such data to external entities through the Internet or with other apps. Confidentiality of user data could be breached if there are anomalies in the way sensitive data is handled by an app which is vulnerable or malicious. Existing approaches that detect anomalous sensitive data flows have limitations in terms of accuracy because the definition of *anomalous flows* may differ for different apps with different functionalities; it is normal for "Health" apps to share heart rate information through the Internet but is anomalous for "Travel" apps.

In this paper, we propose a novel approach to detect anomalous sensitive data flows in Android apps, with improved accuracy. To achieve this objective, we first group *trusted* apps according to the topics inferred from their functional descriptions. We then learn sensitive information flows with respect to each group of trusted apps. For a given app under analysis, anomalies are identified by comparing sensitive information flows in the app against those flows learned from trusted apps grouped under the same topic. In the evaluation, information flow is learned from 11,796 trusted apps. We then checked for anomalies in 596 new (benign) apps and identified 2 previously-unknown vulnerable apps related to anomalous flows. We also analyzed 18 malware apps and found anomalies in 6 of them.

1. INTRODUCTION

Android applications (apps) are often granted access to users' privacy- and security-sensitive information such as GPS position, phone contacts, camera, microphone, training log, and heart rate. Apps need such sensitive data to implement their functionalities and provide rich user experiences. For instance, accurate GPS position is needed to navigate users to their destinations, phone contact is needed to implement messaging and chat functionalities, and heart rate frequency is important to accurately monitor training improvements.

Often, to provide services, apps may also need to exchange data with other apps in the same smartphone or externally with a remote server. For instance, a camera app may share a picture with a multimedia messaging app for sending it to a friend. The messaging app, in turn, may send the full contacts list from the phone directory to a remote server in order to identify which contacts are registered to the messaging service so that they can be shown as possible destinations.

As such, sensitive information may *legitimately* be propagated via message exchanges among apps or to remote servers. On the other hand, sensitive information might be exposed unintentionally by defective/vulnerable apps or intentionally by malicious apps (malware), which threatens the security and privacy of end users. Existing literature on information leak in smartphone apps tend to overlook the difference between legitimate data flows and illegitimate ones. Whenever information flow from a sensitive source to a sensitive sink is detected, either statically [23], [20], [19, 15, 3], [22], [17], [12] or dynamically [8], it is reported as potentially problematic.

In this paper, we address the problem of detecting anomalous information flows with improved accuracy by classifying cases of information flows as either *normal* or *anomalous* according to a reference information flow model. More specifically, we build a model of sensitive information flows based on the following features:

- Data source: the provenance of the sensitive data that is being propagated;
- Data sink: the destination where the data is flowing to; and
- App topic: the declared functionalities of the app according to its description.

Data source and *data sink* features are used to reflect information flows from sensitive sources to sinks and summarize how sensitive data is handled by an app. However, these features are not expressive enough to build an accurate model. In fact, distinct apps might have very different functionalities. What is considered legitimate of a particular set of apps (e.g., sharing contacts for a messaging app) can be considered a malicious behavior for other apps (e.g., a piece of malware that steals contacts, to be later used by spammers). An accurate model should also take into consideration the main functionalities that is declared by an app (in our case the *App topic*). One should classify an app as anomalous

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden
 ISBN 978-1-4503-5712-8/18/05…$15.00
DOI: https://doi.org/10.1145/3197231.3197238

only when it exhibits sensitive information flows that are not consistent with its declared functionalities. This characteristic, which makes an app anomalous, is captured by the *App topic* feature.

In summary, our approach focuses on detecting apps that are anomalous in terms of information flows compared to other apps with similar functionalities. Such an approach would be useful for various stakeholders. For example, market owners (e.g., Google) can focus on performing more complex and expensive security analysis only on those cases that are reported as anomalous, before publishing them. If such information is available to end users, they could also make informed decision of whether or not to install the anomalous app. For example, when the user installs an app, a warning stating that this particular app sends contact information through the Internet differently from other apps with similar functionalities (as demonstrated in the tool website). In the context of BYOD (bring your own device) where employees use their own device to connect to the secure corporate network, a security analyst might benefit from this approach to emphasis manual analysis on those anomalous flows that might compromise the confidentiality of corporate data stored in the devices.

The specific contributions of this paper are:

- An automated, fast approach for detecting anomalous flows of sensitive information in Android apps through a seamless combination of static analysis, natural language processing, model inference, and classification techniques;
- The implementation of the proposed approach in a tool called *AnFlo* which is publicly available[1]; and
- An extensive empirical evaluation of our approach based on 596 subject apps, which assesses the accuracy and runtime performance of anomalous information flow detection. We detected 2 previous-unknown vulnerable apps related to anomalous flows. We also analyzed 18 malware apps and found anomalies in 6 of them.

The rest of the paper is organized as follows. Section 2 motivates this work. Section 3 compares our work with literature. Section 4 first gives an overview of our approach and then explains the steps in details. Section 5 evaluates our approach. Section 6 concludes the paper.

2. MOTIVATION

To implement their services, apps may access sensitive data. It is important that application code handling such data follows secure coding guidelines to protect user privacy and security. However, fast time-to-market pressure often pushes developers to implement data handling code quickly without considering security implications and release apps without proper testing. As a result, apps might contain *defects* that leak sensitive data unintentionally. They may also contain *security vulnerabilities* such as permission redelegation vulnerabilities [9], which could be exploited by malicious apps installed on the same device to steal sensitive data. Sensitive data could also be intentionally misused by *malicious* apps. Malicious apps such as malware and spyware often implement hidden functionalities not declared in their functional descriptions. For example, a malicious app may declare only entertainment features (e.g., games) in its description, but it steals user data or subscribes to paid services without the knowledge and consent of the user.

Defective, vulnerable, and malicious apps all share the same pattern, i.e., they (either intentionally or unintentionally) deal with sensitive data in an anomalous way, i.e., they behave differently in terms of dealing with sensitive data compared to other apps that state similar functionalities. Therefore, novel approaches should focus on detecting anomalies in sensitive data flows, caused by mismatches between expected flows (observed in benign and correct apps) and actual data flows observed in the app under analysis. However, the comparison should be only against *similar* apps that offer similar functionalities. For instance, messaging apps are expected to read information from phone contact list but they are not expected to use GPS position.

These observations motivate our proposed approach.

3. RELATED WORK

The approaches proposed in Mudflow [5] and Chabada [10] are closely related to ours. Mudflow [5] is a tool for malware detection based on sensitive information flow. Similar to our approach, they rely on static taint analysis to detect flows of sensitive data towards potential leaks. Then, these flows are used to train a $\nu-$SVM one-class classifier and later classify new apps. While we also use static analysis, the main difference is that we consider the dominant topic inferred from app description as an important feature for the classification. Our empirical evaluation shows that dominant topics are fundamental to achieve a higher accuracy in anomaly detection. Moreover, our approach not only focuses on detecting malware, but also focuses on vulnerable and defective apps. Lastly, while Mudflow applies intra-component static analysis, we use inter-component analysis for covering flows across components.

Chabada [10] is a tool to find apps whose descriptions differ from their implementations. While we apply similar techniques in terms of natural language processing of apps descriptions, the goals differ. The goal of their approach is to find anomalous apps among the apps in the wild based on the inconsistencies between the advertised apps descriptions and their actual behaviors. By contrast, our approach specifically targets at identifying anomalies in the flow of sensitive information in the app code. More specifically, Chabada only identifies calls to sensitive APIs to characterize benign and anomalous apps. By contrast, we consider data flows from sensitive data sources to sensitive sinks.

Information leak in mobile apps is a widespread security problem. Many approaches that deal with this security problem are related to ours. Information flow in mobile apps is analysed either statically [23], [20], [19], [22], [17], [15], [12] or dynamically [8], to detect disclosure of sensible information. 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 privileged information could potentially leave the app through one of the sinks. In the following, we discuss some of these approaches and then explain the major differences.

Amandroid [23] has been proposed to detect privacy data leaks due to inter-component communication (ICC) in Android apps. Epicc [20] identifies ICC connection points by

[1]Tool and dataset available at http://selab.fbk.eu/anflo/

using inter-procedural data-flow and points-to analysis. IC3 [19] improves Epicc by resolving targets and values used in ICC. Tsutano et al. [22] propose JITANA to analyze interacting apps. It works in a similar fashion as Epicc but instead of combing apps for analysis, it uses static class loader which allows it to analyze large number of interacting apps. IccTA [15] attempts to improve static taint analysis of Android apps in ICC by using IC3 to resolve ICC targets and by modeling the life-cycle and callback methods. DidFail [12] attempts to detect data leaks between activities through implicit intents. It does not consider other components and explicit intents. Grace et al. [11] 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.

TaintDroid [8] 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. A complementary approach is based on static analysis [17], where a type system is implemented to track security levels. It detects violations when privileged information could potentially leave the app through a sink.

Similar to our approach, the above-mentioned approaches apply static analysis techniques on mobile code to detect information flows from sources to sinks. However, our approach does not report all sensitive data flows into sinks as information leak problems because they might be intended behaviors of the app. Our approach classifies information flows in an app as *anomalous* only when they deviate from normal behaviors of other similar apps. In addition, it detects not only cases of information leaks, but also cases of anomalies in data flows that might reveal security defects, such as permission re-delegation vulnerabilities.

Other closely related work is about detecting permission re-delegation vulnerabilities in apps. Felt et al. [9] presented the permission re-delegation problems, and their approach detects them whenever there exists a path from a public entry point to a privileged API call. Chin et al. [7] and Lu et al. [16] also detect permission re-delegation vulnerabilities. However, as acknowledged by Felt et al. and Chin et al., their approaches cannot differentiate between legitimate and illegitimate permission re-delegation behaviors.

Zhang et al. [24] proposed Appsealer, a runtime patch to mitigate permission re-delegation problem. They perform static data flow analysis to determine sensitive data flows from sources to sinks and apply a patch before the invocations of privileged APIs such that the app alerts the user of potential permission re-delegation attacks and requests the user's authorization to continue. This is an alternative way of distinguishing normal behaviors and abnormal ones by relying on the user. Lee et al. [14] also proposed a similar approach but they extended the Android framework to track ICC vulnerabilities instead of patching the app. Instead of relying on the user, who might not be aware of security implications, we resort to a model that reflects normal information flow behaviors to detect anomalies in the flow of sensitive information.

4. ANOMALOUS INFORMATION FLOW DETECTION

4.1 Overview

The overview of our approach is shown in Figure 1. It has two main phases — *Learning* and *Classification*. The input to the *learning phase* is a set of apps that are trusted to be benign and correct in the way sensitive data is handled (we shall denote them as *trusted apps*). It has two sub-steps — feature extraction and model inference. In the feature extraction step, (i) topics that best characterize the trusted apps are inferred using natural language processing (NLP) techniques and (ii) information flows from sensitive sources to sinks in the trusted apps are identified using static taint analysis. In the model inference step, we build sensitive information model that characterizes information flows regarding each topic.

These models and a given app under analysis (we shall denote it as *AUA*) are the inputs to the *classification phase*. In this phase, basically, the dominant topic of the AUA is first identified to determine the relevant sensitive information flow model. Then, if the AUA contains any information flow that violates that model, i.e., is not consistent with the common flows characterized by the model, it is flagged as *anomalous*. Otherwise, it is flagged as *normal*.

We implemented this approach in our tool *AnFlo* to automate the detection of anomalous information flows. However, a security analyst is required to further inspect those anomalous flows and determine whether or not the flows could actually lead to serious vulnerabilities such as information leakage issues.

4.2 Feature Extraction

4.2.1 Topic Analysis

In this step, we analyze the functionalities declared by the trusted apps. Using NLP, we extract topics from the app descriptions available at the official App Store, where apps developer declare their intended functionalities. We fist apply *data pre-processing* to cleanse app descriptions and then we perform *topic discovery* on the pre-processed descriptions to extract the topics that are likely to be associated with.

App descriptions pre-processing. To apply NLP on app descriptions, we need one common language across all apps. Therefore, firstly apps with no English description are filtered. We use Google's Compact Language Detector[2] to detect English language in app descriptions.

Next, *stopwords* in app descriptions are removed. They are words that do not contribute to topic discovery, such as "a", "after", "is", "in", "as", "very", etc[3]. Subsequently, the Stanford CoreNLP lemmatizer[4] is used to apply lemmatization, which basically abstracts the words having similar meanings 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, stemming [21] is applied to reduce words to their radix. Different forms of a word such as "travel", "traveling", "travels", and "traveler" are replaced by their common base form "travel".

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

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

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

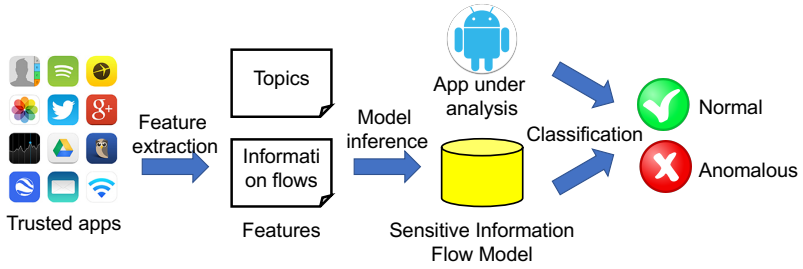


Figure 1: Overview of the approach.

Topics discovery. Topics representative of a given pre-processed app description are identified using the Latent Dirichlet Allocation (LDA) technique [6], implemented in a tool called Mallet [18]. 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. The output of LDA is a list of topics, each of them with its corresponding probability. The topic with the highest probability is labeled as the *dominant topic* for its associated app.

To illustrate, Figure 2 shows the functional description of an app called *BestTravel*, and the resulting output after performing pre-processing and topics discovery on the description. "Travel" is the dominant topic, the one with the highest probability of 70%. Then, the topics "Communication", "Finance", and "Photography" have the 15%, 10%, and 5% probabilities, respectively, of being the functionalities that the app declares to provide.

> The ultimate and most convenient way of traveling. Use BestTravel while on the move, to find restaurants (including pictures and prices), local transportation schedule, ATM machines and much more.

App name	Travel	Communication	Finance	Photography
BestTravel	70%	15%	10%	5%

Figure 2: Example of app description and topic analysis result.

Note that we did not consider Google Play categories as topics even though apps are grouped under those categories in Google Play. This is because recent studies [1, 10] have reported that NLP-based topic analysis on app descriptions produces more cohesive clusters of apps than those apps grouped under Google Play categories.

4.2.2 Static Analysis

Sensitive information flows in the trusted apps are extracted using static taint analysis. Taint analysis is an instance of flow-analysis technique, which tags program data with labels that describe its provenance and propagates these tags through control and data dependencies. A different label is used for each distinct source of data. Tags are propagated from the operand(s) in the right-hand side of an assignment (uses) to the variable assigned in the left-hand side of the assignment (definition). The output of taint analysis is information flows, i.e., what data of which provenances (*sources*) are accessed at what program operations, e.g., on channels that may leak sensitive information (*sinks*).

Our analysis focuses on the flows of sensitive information into sensitive program operations, i.e., our taint analysis generates tags at API calls that read sensitive information (e.g. GPS and phone contacts) and traces the propagation of tags into API calls that perform sensitive operations such as sending messages and Bluetooth packets. These sensitive APIs usually belong to dangerous permission group and hence, the APIs that we analyze are those privileged APIs that require to be specifically granted by the end user. Sources and sinks are the privileged APIs available from PScout [4]. The APIs that we analyze also include those APIs that enable Inter Process Communication (IPC) mechanism of Android because they can be used to exchange data among apps installed on the same device.

As a result, our taint analysis generates a list of (*source* $\rightarrow$ *sink*) pairs, where each pair represents the flow of sensitive data originating from a source into a sink.

APIs (both for sources and for sinks) are grouped according to the special *permission* required to run them. For example, all the network related sink functions, such as `openConnection()`, `connect()` and `getContent()` are all modeled as *Internet* sinks, because they all require the `INTERNET` permission to be executed.

Figure 3 shows the static taint analysis result on the "BestTravel" running example app from Figure 2. It generates two (*source* $\rightarrow$ *sink*) pairs that correspond to two sensitive information flows. In the first flow, data read from the GPS is propagated through the program until it reaches a statement where it is sent over the network. In the second flow, data from the phone contacts is used to compose a text message.

App: BestTravel		
GPS	$\rightarrow$	Internet
Contacts	$\rightarrow$	SMS

Figure 3: Example of static analysis result.

Our tool, *AnFlo*, runs on compiled byte-code of apps to perform the above static taint analysis. It relies on two existing tools — IC3 [19] and IccTA [15]. Android apps are usually composed of several components. Therefore, to precisely extract inter-component information flows, we need to analyze the links among components. *AnFlo* uses IC3 to resolve the target components when a flow is inter-component. IC3 uses a solver to infer all possible values of complex objects in an inter-procedural, flow- and context-sensitive manner. Once inter-component links are inferred, *AnFlo* uses an inter-component data-flow analysis tool called IccTA to perform static taint analysis. We customized IccTA to produce flows in a format as presented in Figure 3 and paths in a more verbose format to facilitate manual checks.

4.3 Model Inference

When results of topic analysis and of static analysis are available for all the trusted apps, they are used to build the *Sensitive Information Flow Model*. Such a model is a matrix with sensitive information sources in its rows and sinks in its columns, as shown in Figure 4.

Firstly, apps with the same dominant topic are grouped together[5], to build a sensitive information flow model corresponding to that specific topic. Each group is labeled with the dominant topic. Next, each cell of the matrix is filled with a number, representing the number of apps in this group having the corresponding ($source \rightarrow sink$) pair.

Figure 4 shows a sample sensitive information model regarding the topic "Travel". There are 36 distinct flows in the apps grouped under this dominant topic. The matrix shows that there are ten apps containing *GPS position* flowing through the Internet (one of them being the *BestTravel* app, see Figure 3); eight apps through text messages and three apps through Bluetooth. Similarly, the matrix shows that *contacts* information flows through SMS in seven apps and through Bluetooth in eight apps.

Topic: "Travel"	**Sinks**		
Sources	Internet	SMS	Bluetooth
GPS	10	8	3
Contacts	0	7	8

Figure 4: Example of sensitive information flow model.

From this model, we can observe that for Travel apps it is quite *common* to share the user's position via Internet and SMS. However, it is quite *uncommon* to share the position data via Bluetooth since it happened only in three cases. Likewise, the phone contacts are *commonly* shared through text messages and Bluetooth but not through Internet.

To provide a formal and operative definition of *common* and *uncommon* flows, we compute a threshold denoted as τ. Flows that occur more than or equal to τ are considered as *common*; flows that never occur or that occur fewer than τ are considered as *uncommon* regarding this topic.

Although our model assumes or trusts that the *trusted* apps are benign and correct, it is possible that some of them may contain defects, vulnerabilities or malware. This problem is addressed by classifying those flows occurring less than the threshold τ as uncommon, i.e., our approach tolerates the presence of some anomalous flows in the reference model since these flows would still be regarded as uncommon. Hence, our approach works as long as the *majority* of the trusted apps are truly trustworthy.

To compute this threshold, we adopt the box-plot approach proposed by Laurikkala et al. [13], considering only flows occurring in the model, i.e., we consider only values greater than zero. τ is computed in the same way as drawing outlier dots in boxplots. It is the lower quartile (25th percentile) minus the step, where the step is 1.5 times the difference between the upper quartile (75th percentile) and the lower quartile (25th percentile). It should be noted that τ is not trivially the lower quartile; otherwise 25% of the apps would be *outliers* by construction. The threshold is lower, i.e., it is the lower quartile minus the step. Therefore, there is no fixed amount of outliers. Outliers could be few or many depending on the distribution of data. Outliers would only be those cases that are really different from the majority of the training data points.

In the example regarding topic "Travel" in Figure 4, the threshold is computed considering only the five values that are > 0. The value for the threshold is $\tau_{Travel} = 7$. It means that GPS data sent through Internet (GPS $\rightarrow$ Internet) or text messages (GPS $\rightarrow$ SMS) are *common* for traveling apps. Conversely, even though there are three trusted apps which send GPS data through Bluetooth (GPS $\rightarrow$ Bluetooth), there are too few cases to be considered common, and this sensitive information flow will be considered *uncommon* in the model. Likewise, phone contacts are *commonly* sent through text messages and Bluetooth, but it is *uncommon* for them to be sent through the Internet, since this *never* occurs in the trusted apps.

4.4 Classification

After the Sensitive Information Flow Models are built on trusted apps, they can be used to classify a new AUA. First of all, features must be extracted from the AUA. The features are the topics associated with the app description and the sensitive information flows in the app. As in Section 4.2.1, first data pre-processing is performed on the app description of the AUA. Then, topics and their probabilities are inferred from the pre-processed description using the Mallet tool. Among all the topics, we consider only the *dominant topic*, the one with the highest probability, because it is the topic that most characterizes this app. We then obtain the Sensitive Information Flow Model associated with this dominant topic.

To ensure the availability of the Sensitive Information Flow Model, the Mallet tool is configured with the list of topics for which the Models are already built on the trusted apps. And given an app description, the Mallet tool only generates topics from this list. The more diverse trusted apps we analyze, the more complete list of models we expect to build.

For example, Figure 5(a) shows the topics inferred from the description of a sample AUA "TripOrganizer". The topic "Travel" is highlighted in bold to denote that it is the dominant topic.

Next, sensitive information flows in the AUA are extracted as described in Section 4.2.2. The extracted flows are then compared against the flows in the model associated with the dominant topic. If the AUA contains only flows that are *common* according to the model, the app is considered consistent with the model. If the app contains a flow that is not present in the model or a flow that is present but is *uncommon* according to the model, the flow and thus, the app is classified as anomalous. Anomalous flows require further manual inspection by a security analyst, because they could be due to defects, vulnerabilities, or malicious intentions.

For example, Figure 5(b) shows three sensitive information flows extracted from "TripOrganizer" app. Since the dominant topic for this app is "Travel", these flows can be checked against the model associated with this topic shown in Figure 4. Regarding this model, earlier, we computed that the threshold is $\tau_{Travel} = 7$ and the flow (Contacts $\rightarrow$ SMS) is common (see Section 4.3). Therefore, flow 1 ob-

[5]We also experimented with a more elaborated model that considers multiple topics and their probabilities instead of just the dominant topic for grouping the apps. However, since their detection accuracy did not improve, we opted for the simplest model, with just the dominant topic.

served in "TripOrganizer" (Figure 5(b)) is consistent with the model. However, flow 2 (Contacts → Internet) and flow 3 (GPS → Bluetooth), highlighted in bold in Figure 5(b), are uncommon according to the model. As a result, the AUA "TripOrganizer" is classified as *anomalous*.

App name	**Travel**	Books	Tools	Game
TripOrganizer	**78%**	11%	4%	7%

(a) Topics classification

App: TripOrganizer			
1:	Contacts	→	SMS
2:	**Contacts**	→	**Internet**
3:	**GPS**	→	**Bluetooth**

(b) Sensitive information flows

Figure 5: Classification of the app under analysis.

5. EMPIRICAL ASSESSMENT

In this section, we evaluate the usefulness of our approach and report the results. We assess our approach by answering the following research questions:

- $\mathbf{RQ}_{Vul}$: Is *AnFlo* useful for identifying *vulnerable apps* containing anomalous information flows?
- $\mathbf{RQ}_{Time}$: How *long* does *AnFlo* take to classify apps?
- $\mathbf{RQ}_{Topics}$: Is the *topic* feature really needed to detect anomalous flows?
- $\mathbf{RQ}_{Cat}$: Can app-store *categories* be used instead of *topics* to learn an accurate Sensitive Information Flow Model?
- $\mathbf{RQ}_{Mal}$: Is *AnFlo* useful for identifying *malicious* apps?

The first research question $\mathbf{RQ}_{Vul}$ investigates the result of *AnFlo*, whether it is useful for detecting anomalies in vulnerable apps that, for example, may leak sensitive information. $\mathbf{RQ}_{Time}$ investigates the cost of using our approach in terms of the time taken to analyze a given AUA. A short analysis time is essential for tool adoption in a real production environment.

Then, in the next two research questions, we investigate the role of *topics* as a feature for building the Sensitive Information Flow Models. $\mathbf{RQ}_{Topics}$ investigates the absolute contribution of topics, by learning the Sensitive Information Flow Model without considering the topics and by comparing its performance with that of our original model. To answer $\mathbf{RQ}_{Cat}$, we replace topics with the *categories* defined in the official market, and we compare the performance of this new model with that of our original model.

Finally, the last research question $\mathbf{RQ}_{Mal}$ investigates the usefulness of *AnFlo* in detecting malware based on anomalies in sensitive information flows.

5.1 Benchmarks and Experimental Settings

5.1.1 Trusted Apps

AnFlo needs a set of trusted apps to learn what is the *normal* behavior for "correct and benign" apps. We defined the following guidelines to collect trusted apps: (i) apps that come from the official Google Play Store (so they are scrutinized and checked by the store maintainer) and (ii) apps that are very popular (so they are widely used and reviewed by a large community of end users and programming mistakes are quickly notified and patched).

At the time of crawling the Google Play Store, it had 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 very short descriptions (less than 10 words). Eventually, we are left with 11,796 apps for building references models.

Additionally, we measured if these apps were actively maintained by looking at the date of the last update. 70% of the apps were last updated in the past 6 months before the Play Store was crawled, while 32% of the apps were last updated within the same month of the crawling. This supports the claim that the trusted apps are well maintained.

The fact that the trusted apps we use are suggested and endorsed by the official store, and that they collected good end-user feedback allows us to assume that the apps are of high quality and do not contain many security problems. Nevertheless, as explained in Section 4.3, our approach is robust against the inclusion of a small number of anomalous apps in the training set since we adopt a threshold to classify anomalous information flows.

5.1.2 Subject Benign Apps

AnFlo works on compiled apps and, therefore the availability of source code is not a requirement for the analysis. However, for this experiment sake, we opted for open source projects, which enable us to inspect the source code and establish the *ground truth*.

The F-Droid repository[6] represents an ideal setting for our experimentation because (i) it includes real world apps that are also popular in the Google Play Store, and (ii) apps can be downloaded with their source code for manual verification of the vulnerability reports delivered by *AnFlo*.

The F-Droid repository was crawled in July 2017 for apps that meet our criteria. Among all the apps available in this repository, we used only those apps that are also available in the Google Play Store, whose descriptions meet our selection criteria (i.e., description is in English and it is longer than 10 words). Eventually, our experimental set of benign apps consists of 596 AUAs.

5.1.3 Subject Malicious Apps

To investigate if *AnFlo* can identify malware, we need a set of malicious apps with their *declared* functional descriptions. Malicious apps are usually repackaged versions of popular (benign) apps, injected with malicious code (Trojanized); hence the descriptions of those popular apps they disguise as can be considered as their app descriptions. Hence, by identifying the original versions of these malicious apps in the Google Play Store, we obtain their declared functional descriptions.

We consider the malicious apps from the Drebin malware dataset [2], which consists of 5,560 samples that have been collected in the period of August 2010 to October 2012. We randomly sampled 560 apps from this dataset. For each malicious app, we performed static analysis to extract the

[6] `http://f-droid.org/`

package name, an identifier used by Android and by the official store to distinguish Android apps[7]. We queried the official Google Play market for the original apps, by searching for those having the same package name. Among our sampled repackaged malicious apps, we found 20 of the apps in the official market with the same package name. We analyzed their descriptions and found that only 18 of them have English descriptions. We therefore performed static taint analysis on these 18 malware samples, for which we found their "host" apps in the official market. Our static analysis crashed on 6 cases. Therefore, our experimental set of malicious apps consists of 12 AUAs.

5.2 Results

5.2.1 Detecting Vulnerable Apps

Firstly, *AnFlo* was used to perform static taint analysis on the 11,796 trusted apps and topic analysis on their descriptions from the official Play Store. It then learns the Sensitive Information Flow Models based on the dominant topics and extracted flows as described in Section 4.3. Then, the AUAs from the F-Droid repository (Section 5.1.2) have been classified based on the Sensitive Information Flow Models.

Out of 596 AUAs, static taint analysis reported 76 apps to contain flows of sensitive information that reach sinks, for a total of 1428 flows. These flows map to 147 distinct source-sink pairs. Out of these 76 apps, 14 AUAs are classified as *anomalous*. Table 1 shows the analysis results reported by *AnFlo*. The first column presents the name of the app. The second column presents the app's dominant topic. The third and fourth columns present the source of sensitive data and the sink identified by static taint analysis, respectively. As shown in Table 1, in total *AnFlo* reported 25 anomalous flows in these apps. We manually inspected the source code available from the repository to determine if these anomalous flows were due to programming defects or vulnerabilities. Two apps are found to be vulnerable (highlighted in boldface in Table 1), they are *com.matoski.adbm* and *com.mschlauch.comfortreader*.

com.matoski.adbm is a utility app for managing the ADB debugging interface. The anomalous flow involves data from the WiFi configuration that leak to other apps through the Inter Process Communication. Among other information that may leak, the SSID data, which identifies the network to which the device is connected to, can be used to infer the user position and threaten the end user privacy. Hence, this programming defect leads to information leakage vulnerability that requires corrective maintenance. We reported this vulnerability to the app owners on their issue tracker.

com.mschlauch.comfortreader is a book reader app, with an anomalous flow of data from IPC to the Internet. Manual inspection revealed that this anomalous flow results from a permission re-delegation vulnerability because data coming from another app is used, without sanitization, for opening a data stream. If a malicious app that does not have the permission to use the Internet passes a URL that contains sensitive privacy data (e.g., GPS coordinates), then the app could be used to leak information. We reported this vulnerability to the app developers.

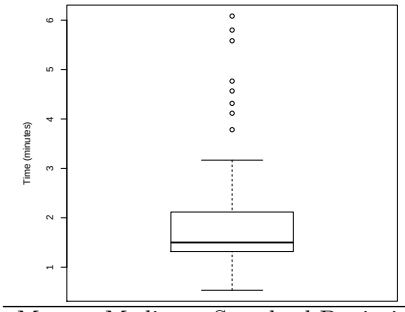
Regarding the other 12 AUAs, even though they contain anomalous flows compared to trusted apps, manual inspection revealed that they are neither defective nor vulnerable. For example, some apps contain anomalous flows that involves IPC. Since data may come from other apps via IPC (source) or may flow to other apps via IPC (sink), such flows are considered dangerous in general. However, in these 12 apps, when IPC is a source (e.g., in `com.alfray.timeriffic`), data is either validated/sanitized before used in the sink or used in a way that do not threaten security. On the other hand, when IPC is a sink (e.g., in `com.dozingcatsoftware.asciicam`), the destination is always a component in the same app, so the flows are not actually dangerous.

Since *AnFlo* helped us detect 2 vulnerable apps containing anomalous information flows, we can answer $\mathbf{RQ}_{Vul}$ by stating that *AnFlo* is useful for identifying vulnerabilities related to anomalous information flows.

5.2.2 Classification Time

To investigate $\mathbf{RQ}_{Time}$, we analyze the time required to classify the AUAs. We instrumented the analysis script with the Linux *date* utility to log the time (in seconds) before starting the analysis and at its conclusion. Their difference is the amount of time spent in the computation. The experiment was run on a multi-core cluster, specifically designed to let a process run without sharing memory or computing resources with other processes. Thus, we assume that the time measurement is reliable.

Classification time includes the static analysis step to exact data flow, the natural language step to extract topics from description and the comparison with the Sensitive Information Flow Model to check for consistency. Figure 6 reports the boxplot of the time (in minutes) needed to classify the F-Droid apps and the descriptive statistics. On average, an app takes 1.9 minutes to complete the classification and most of the analyses concluded in less than 3 minutes (median = 1.5). Only a few (outliers) cases require longer analysis time.



Mean	Median	Standard Deviation
1.9	1.5	1.07

Figure 6: Boxplot of classification time.

5.2.3 Topics from App Description

We now run another experiment to verify our claim that topics are important features to build an accurate model ($\mathbf{RQ}_{Topics}$). We repeated the same experiment as before, but using only flows as features and without considering topics, to check how much detection accuracy we lose in this way.

We still consider all the trusted apps for learning the reference model, but we only use static analysis data. That

[7]Even if it is easy to obfuscate this piece of information, in our experiment some apps did not rename their package name

Table 1: Anomaly detection based on Sensitive Information Flows Model

App	Topic	Source	Sink
a2dp.Vol	communications utility	Bluetooth Bluetooth Bluetooth admin Broadcast sticky Modify audio settings	IPC Modify audio settings Modify audio settingss Modify audio settingss Modify audio settingss
com.alfray.timeriffic	tools and utility	IPC	Write settingss
com.dozingcatsoftware.asciicam	photo editors	Camera	IPCs
com.matoski.adbm	**utility**	**Access WiFi state**	**IPC**
com.mschlauch.comfortreader	**books & readers**	**IPC**	**Internet**
com.newsblur	productivity	IPC	Access network states
com.Pau.ImapNotes2	documents manager	Authenticate accounts Authenticate accounts Get accounts Get accounts Get accounts	IPCs Authenticate accountss Authenticate accountss Get accountss Manage accountss
com.thibaudperso.sonyCamera	tools & utility	IPC NFC	Change WiFi states NFCs
fr.neamar.kiss	security tools	IPC	Access WiFi states
fr.ybo.transportsrennes	utility	Access coarse location Access fine location	Internets Internets
mobi.boilr.boilr	financial	IPC Wake lock	Wake lock Wake lock
org.smc.inputmethod.indic	languages learning	IPC	Vibrate
pw.thedrhax.mosmetro	books & readers	Access WiFi state	Change WiFi state
se.anyro.Nfc_reader	communications utility	IPC	NFC

is, we do not create a separate matrix for each topic; instead we create one big single matrix with sources and sinks for all the apps. This Sensitive Information Flow Model is then used to classify F-Droid apps and the results are shown in Table 2. As we can see, only four apps are detected as *anomalous* by this second approach, and all of them were already detected by our original, proposed approach. Manual inspection revealed that all of them are not vulnerable.

This suggests that topic is a very important feature to learn reference models in order to detect a larger amount of anomalous apps. In fact, when topics are not considered and all the apps are grouped together regardless of their topics, we observe a *smoothing* effect. Differences among apps become less relevant to detect anomalies. While in the previous model, an app was compared only against those apps grouped under the same topic. Here, an app is compared to all the trusted apps. Without topic as a feature, our model loses the ability to capture the characteristics of distinct groups and, thus, the ability to detect deviations from them.

5.2.4 Play Store Categories

To investigate $\mathbf{RQ}_{Cat}$, instead of grouping trusted apps based on topics, we group them according to their app categories as determined by the official Google Play Store. First of all we split trusted apps into groups based on the market category they belong to[8]. We then use static analysis information about flows to build a separate source-sink matrix per each category. Eventually we compute thresholds to complete the model.

We then classify each AUA from F-Droid by comparing it with the model of the corresponding market category. The classification results are reported in Table 3. Ten apps are reported as containing anomalous flows and most of them were also detected by our original, proposed approach (Table 1). Two apps reported by this approach were not reported by our proposed approach, which are *com.angrydoughnuts.android.alarmclock* and *com.futurice.android.reservator*. However, they are neither the cases of vulnerabilities nor malicious behaviors. Only one flow detected by this approach is a case of vulnerability, namely *com.matoski.adbm*, highlighted in boldface, which was also detected by our proposed approach. Hence, this result supports our design decision of using topics.

5.2.5 Comparison of the Models

Table 4 summarizes the result of the models comparison. The first model (first row) considers both data flows and description topics as features. Even though this approach reported the largest number of false positives (12 apps, 'FP' column), we were able to detect 2 vulnerabilities ('Vuln.' column) by tracing the anomalies reported by this approach. It also detected 5 additional anomalous apps that other approaches did not detect ('Unique' column).

The second model (second row) considers only data flows as a feature. Even though the number of false positives drops to 4, we were not able to detect any vulnerability by tracing the anomalies reported by this approach. This result suggests that modeling only flows is not enough for detecting vulnerabilities.

When market categories are used instead of description

[8]At the time of the crawling, in Google Play Store, a popular app was assigned to only one category.

Table 2: Anomaly detection using only information flows as a feature (no topic feature)

App	Topic	Source	Sink
a2dp.Vol	Communications utility	Bluetooth	Modify audio settings
		Bluetooth admin	Modify audio settings
		Broadcast sticky	Modify audio settings
		Modify audio settings	Modify audio settings
com.Pau.ImapNotes2	Document manager	Authenticate accounts	IPC
		Authenticate accounts	Authenticate accounts
		Get accounts	Authenticate accounts
		Get accounts	Get accounts
		Get accounts	Manage accounts
com.thibaudperso.sonyCamera	Utility	IPC	Change wifi state
		NFC	NFC
se.anyro.Nfc_reader	Communications utility	IPC	NFC

Table 3: Anomaly detection using Google Play categories as a feature instead of topics

App	Category	Source	Sink
a2dp.Vol	Transportation	Bluetooth	IPC
		Bluetooth	Bluetooth
		Bluetooth	Modify audio settings
		Bluetooth admin	Modify audio settings
		Broadcast sticky	Modify audio settings
		Modify audio settings	Modify audio settings
com.Pau.ImapNotes2	Productivity	Authenticate accounts	IPC
		Authenticate accounts	Authenticate accounts
		Get accounts	Authenticate accounts
		Get accounts	Get accounts
		Get accounts	Manage accounts
com.alfray.timeriffic	Tools	IPC	Write settings
com.angrydoughnuts.android.alarmclock	Tools	Wake lock	Vibrate
com.dozingcatsoftware.asciicam	Photography	Camera	IPC
com.futurice.android.reservator	Business	Get accounts	IPC
com.matoski.adbm	**Tools**	**Access wifi state**	**IPC**
com.thibaudperso.sonyCamera	Media and video	IPC	Change wifi state
		NFC	NFC
mobi.boilr.boilr	Finance	Wake lock	Wake lock
se.anyro.Nfc_reader	Communication	IPC	NFC

Table 4: Summary of model comparison result

RQ	Features	FP	Unique	Vuln.
RQ_{Vul}	Flows + Topics	12	5	2
RQ_{Topics}	Flows	4	0	0
RQ_{Cat}	Flows + Market Cat.	9	2	1

topics (last row), the false positives drops to 9 (25% less compared to our proposed model). It detected 2 additional anomalous apps that other approaches did not detect ('Unique' column). Tracing the anomalies reported by this approach, we detected only one out of the two vulnerabilities that we detected using our proposed approach. This result suggests that topics are more useful than categories for detecting vulnerable apps containing anomalous information flows.

5.2.6 Detecting Malicious Apps

Anomalies in the flow of sensitive data could be due to malicious behaviors as well. The goal of this last experiment is to investigate whether *AnFlo* can be used to identify malware ($\mathbf{RQ}_{Mal}$). To this aim, we use the Sensitive Information Flow Model (learned on the trusted apps) to classify the 18 AUAs from the Drebin malware dataset. Data flow features are extracted using static analysis from these malicious apps. However, static taint analysis crashed on 6 apps because of their heavy obfuscation. Since improving the static taint analysis implementation to work on heavy obfuscated code is out of the scope of this paper, we run the experiment on the remaining 12 apps. Topics are extracted from the descriptions of the original versions of those malware, which are available at the official market store.

The malicious apps have been subject to anomaly detection, based on the three distinct feature sets: (i) flows and topics; (ii) only flows; and (iii) flows and market categories. The classification results are shown in Table 5. The first column reports the malware name (according to ESET-NOD32[9] antivirus) and the second column contains the name of the original app that was repackaged to spread the malware. The remaining three columns report the results of malware detection by the three models based on different sets of features: a tick mark ("✔") means that the

[9]https://www.eset.com/

Table 5: Results on malicious apps

Malware Name	Repackage App Name	Flows + Topics	Flows only	Flows + Market Cat.
PJApps	com.appspot.swisscodemonkeys.steam	✔	✔	✔
DroidKungFu (variant 1)	com.gp.jaro (variant)	✔	✔	✔
DroidKungFu (variant 2)	com.gp.jaro (variant)	✔	✖	✔
Spy.GoldDream	com.rechild.advancedtaskkiller	✔	✔	✔
Anserver	com.sohu.blog.lzn1007.WatermelonProber	✔	✖	✔
TrojanSMS.Agent	org.baole.app.blacklistpro	✔	✔	✔

model correctly detected the app as anomalous, while a cross ("✖") means no anomaly detected.

While the model based on topics and the model based on market categories classified the same 6 AUAs as malicious, the model based on only flows classified only 4 AUAs as malicious.

All the malware except `TrojanSMS.Agent` are the cases of privacy sensitive information leaks such as device ID, phone number, e-mail or GPS coordinate, being sent over the network or via SMS. One typical malicious behavior is observed in `Spy.GoldDream`. In this case, after querying the list of installed packages (sensitive data source), the malware attempts to kill selected background processes (sensitive sink). This is a typical malicious behavior observed in malware that tries to avoid detection by stopping security products such as antiviruses. Botnet behavior is observed in `DroidKunFu`. A command and control (C&C) server command is consulted (sensitive source) before performing privileged actions on the device (sensitive sink).

As shown in Table 5, when only static analysis features are used in the model, two malicious apps are missed. This is because this limited model compares the given AUA against all the trusted apps, instead of only the apps from a specific subset (grouped by the common topic or the same category). A flow that would have been anomalous for the specific topic (or the specific category) might be normal for another topic/category. For example, acquiring GPS coordinate and sending it over the network is common for navigation or transportation apps. However, it is not a common behavior for tools apps, which is the case of the `Anserver` malware.

The remaining 6 apps in the dataset were consistently classified as not-anomalous by all the models. These false negatives are mainly due to the malicious behaviors not related to sensitive information flows, such as dialing calls in the background or blocking messages. Another reason is due to the obfuscation by malware to hide the sensitive information flows. Static analysis inherently cannot handle obfuscation.

5.3 Limitation and Discussion

In the following, we discuss some of the limitations of our approach and of its experimental validation. The most prominent limitation to adopt our approach is the availability of trusted apps to build the model of sensitive information flows. In our experimental validation, we trusted top ranked popular apps from the official app store, but we have no guarantee that they are all immune from vulnerabilities and from malware content. However, as explained in Section 4.3, our approach is quite robust with respect to the inclusion of a small number of defective, vulnerable, or malicious apps in the training set, as long as the majority of the training apps are benign and correct. This is because we use a threshold-based approach that models flows common to a large set of apps. Thus, vulnerable flows occurring on few training apps are not learnt as normal in the model and they would be classified as *anomalous* when observed in a given AUA.

A flow classified as *anomalous* by our model needs further manual analysis to check if the anomaly is a vulnerability, a malicious behavior or is safe. Manual inspection could be an expensive task that might delay the delivery of the software product. However, in our experimental validation, manual filtering on the experimental result took quite short time, on average 30 minutes per app. Considering that the code of the app to review was new to us, we expect a shorter manual filtering phase for a developer who is quite familiar with the code of her/his app. All in all, manual effort required to manual filter results of the automated tool seems to be compatible with the fast time-to-market pressure of smart phone apps.

When building sensitive information flow models, we also considered grouping of apps by using clustering technique based on the topics distribution, instead of grouping based on the dominant topic alone. But we conducted preliminary experiments using this method and observed that grouping of apps based on dominant topics produce more cohesive groups, i.e., apps that are more similar.

Inherently, it is difficult for static analysis-based approaches including ours to handle obfuscated code. Therefore, if training apps are obfuscated (e.g., to limit reverse engineering attacks), our approach may collect incomplete static information and only build a partial model. And if the AUA is obfuscated, our approach may not detect the anomalies. As future work, we plan to incorporate our approach with dynamic analysis to deal with obfuscation.

6. CONCLUSION

In this paper, we proposed a novel approach to analyze the flows of sensitive information in Android apps. In our approach, trusted apps are first analyzed to extract topics from their descriptions and data flows from their code. Topics and flows are then used to learn Sensitive Information Flow models. We can use these models for analyzing new Android apps to determine whether they contain anomalous information flows. Our experiments show that this approach could detect anomalous flows in vulnerable and malicious apps quite fast.

Acknowledgment

This work has partially been supported by the activity "API Assistant" of the action line Digital Infrastructure of the EIT Digital and the GAUSS national research project, which has been funded by the MIUR under the PRIN 2015 program (Contract 2015KWREMX). The work of L. K. Shar has been partially supported by the National Research Fund, Luxembourg, while the author was affiliated with the University of Luxembourg (INTER/AAL/15/11213850 and INTER/DFG/14/11092585).

7. REFERENCES

[1] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang. 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, pages 38:1–38:10, New York, NY, USA, 2016. ACM.

[2] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 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, pages 259–269, New York, NY, USA, 2014. ACM.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[5] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436, May 2015.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[8] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th Usenix Symposium on Operating Systems Design and Implementation*, 2010.

[9] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th Usenix Security Symposium*, 2011.

[10] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[11] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*. The Internet Society, 2012.

[12] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. 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, pages 1–6, New York, NY, USA, 2014. ACM.

[13] J. Laurikkala, M. Juhola, E. Kentala, N. Lavrac, S. Miksch, and B. Kavsek. Informal identification of outliers in medical data. In *Fifth International Workshop on Intelligent Data Analysis in Medicine and Pharmacology*, volume 1, pages 20–24, 2000.

[14] Y. K. Lee, J. y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic. A sealant for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 312–323, Piscataway, NJ, USA, 2017. IEEE Press.

[15] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, pages 280–291, 2015.

[16] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[17] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *27th Symposium on Applied Computing (SAC): Computer Security Track*, pages 1457–1462, 2012.

[18] A. K. McCallum. Mallet: A machine learning for language toolkit. 2002.

[19] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, May 2015.

[20] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. 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, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[21] M. F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[22] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 324–334, Piscataway, NJ, USA, 2017. IEEE Press.

[23] F. Wei, S. Roy, X. Ou, and Robby. 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, pages 1329–1341, New York, NY, USA, 2014. ACM.

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