
# EVALUATION OF MACHINE LEARNING ALGORITHMS IN THE CATEGORIZATION OF ANDROID API METHODS INTO SOURCES AND SINKS

**By**

***WALBER DE MACEDO RODRIGUES***

**B.Sc. Dissertation**


Federal University of Pernambuco
secgrad@cin.ufpe.br
www.cin.ufpe.br/~secgrad

RECIFE/2018

Resumo

Um programa computa em dados sensíveis e não sensíveis, sendo sensível, qualquer dado que possa identificar um usuário, como GPS, fotos e videos. Esses dados seguem um fluxo específico indo de *data sources* para *data sinks*, *data sources* são métodos ou funções que geram dados sensíveis, *data sinks* são funções que consomem qualquer tipo de dados, sensível ou não. O vazamento de dados acontece quando dados sensíveis chegam sem autorização em *sinks*, para prevenir isso, técnicas de *Flow Enforcement* que garantem que dados sensíveis não cheguem em *sinks* não autorizados. Para isso, esses métodos usam listas, geradas manualmente, de métodos que sejam *sources* sensíveis ou *sinks*, e essa solução é impraticável para grandes APIs como a do Android. Visto isso, uma abordagem usando *machine learning* foi desenvolvida para classificar esses métodos *sources* e *sinks*. O presente trabalho tem como objetivo criar um dataset para avaliar os métodos de classificação para decidir qual é o mais apropriado para esse problema. Para criar o dataset, foram utilizados as APIs Android Level 3 a Level 27, excluindo as de Level 18 e 20, abrangendo a maioria das APIs recentes que foram aplicadas em um extrator de features. Para realizar a classificação, foram amostrados aleatoriamente 30 datasets de treino e teste e usadas as métricas de precisão, revocação, medida F1 e acurácia dos classificadores em cada um dos datasets e feito um teste de hipótese para avaliar se os classificadores são equivalentes.

Abstract

A program computes in either sensitive and non-sensitive data and follows a specific flow, from data sources to data sinks, data sources are methods or functions that create sensitive data, data sinks are functions that consumes any kind of information, sensitive or not. Data leakage happens when sensitive data is sent to unauthorized data sinks, to prevent that, Dynamic and Static Flow Enforcement techniques ensure that sensitive data reaches those sinks. To prevent data leakage, these methods rely on a list of sensitive data sources and data sinks, this list is hand annotated and is practical to be made to a huge API such as Android. With that in mind, a machine learning model is used to classify methods into sources and sinks. The present work intends to extend the previous work creating a dataset to evaluate the most used classification algorithms and define which is the most suitable to this problem. This work has as objective create a dataset of Android methods to evaluate classifiers and decide which one is the most suited to this problem. To create the dataset, we used real Android APIs from Level 3 to Level 27, excluding Levels 18 and 20, embracing all newer APIs that have been used in a feature extractor. To decide which Monolithic or Multi Classifier Systems solves this classification problem, we random sampled 30 dataset into train and test, calculated precision, recall, F1 score and accuracy and used a statistical test to evaluate if the classifiers are equivalent.

List of Figures

List of Tables

Contents

# 1

# Introduction

## 1.1 Motivation

The expression Internet of Things is relatively new, was coined between 1997 and 2000 and consist of devices that sense or actuate in a real environment (MINERVA; BIRU; ROTONDI (2015)). The number of IoT devices has already overcame the living humans on Earth and, by 2020, is estimated to exist about 20 billion of IoT devices connected to the internet (GROWTHENABLER, 2017). It is very common that these devices have a companion app, cloud or mobile based which are used to control and monitor the IoT devices.

This growing popularity come along with major security and privacy issues, hackers are exploiting Android apps to leak user private information or hijacking the IoT device to use in other attacks, such as DDoS (YOUNG, 2018). The information leak is very concerning from a privacy point of view, as a vast majority of user sensitive information is stored in smartphones, GPS location, text messages, photos and even bank data can be stolen if exists an issue in IoT companion apps.

IoT applications consists in a device that can sense and actuate in the real environment which is controlled and monitored using mobile apps or a web server to serve for any kind of purpose, from medical monitoring to smart wearable devices. Depending of the application the devices are designed to last for years with low or no need of maintenance or be portable. To ensure that these devices are hardware constrained, lacking computational power to ensure a longer battery life (RAJ; RAMAN, 2017). This leads to some issues: the needing of newer lightweight security protocols to fit the hardware constrains as ZHANG et al. (2014) points and a reduced set of communications protocols and interfaces to keep low power consumption.

The lack of communications protocols forced the industry to create other devices to act as an IoT hub, if the IoT does not have an independent connection to the internet, hub architectures are common if the device have simpler and short communication protocols, like Bluetooth, RFID, ZigBee and NFC AL-SARAWI et al. (2017). The hub concentrate the communications between the devices and the companion app, this app is a mobile or cloud application that provides monitoring and control for the IoT devices. The real IoT communication architecture is very diverse but a device have few options, it can only send information through the hub to the cloud,

to a smartphone, directly to the cloud or have these three combined using a mixed architecture.

In Figure 1.1 we can observe how the IoT device communicate directly to user devices, the smart devices are represented by a smart bulb and the user devices by a smartphone and a desktop computer. The full arrows express that that the link must exist to have a connection. In Figure 1.2, we have a intermediate device that makes the connection between the device and the user, the hub can aggregate as many IoT devices as intended. The cloud architecture Figure 1.3 consists in a device that is connected to the internet using a router, which sends and receives user actions using the cloud as intermediary. The mixed architecture, Figure 1.4, can have all the previous kind of connections, the user has freedom to send information directly to the device, or use a hub to control the devices with only one application or send remote requests using any cloud provider.

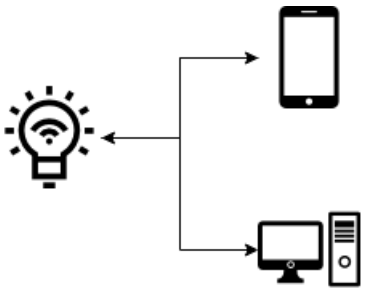

**Figure 1.1:** Direct architecture

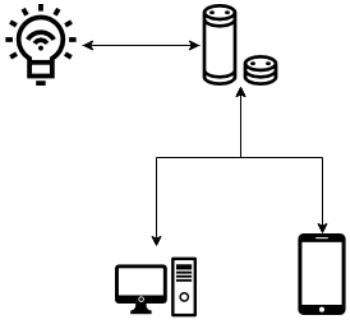

**Figure 1.2:** Hub architecture

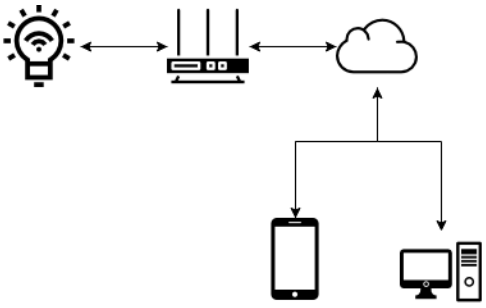

**Figure 1.3:** Cloud based architecture

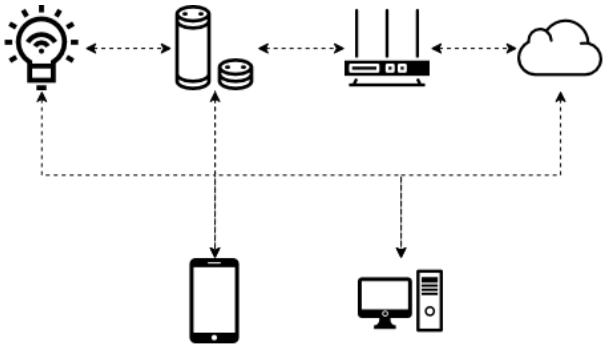

**Figure 1.4:** Mixed architecture

With the increasing popularity of IoT, attackers are exploiting weakness in the architecture, the device itself and the smartphones that hosts the application. Exists many known attacks, from leaking user data as described by YOUNG (2018) to hijacking the device to use in other attacks such as DDoS using botnets (KOLIAS et al., 2017). The information leaking is a major concern from a privacy point of view, this kind of attack can leak any kind of user sensitive data being sent to malicious hosts or unaware to a cloud server.

Sensitive data is any data that can be used to identify the user or any private user information, such as photos, International Mobile Equipment Identity (IMEI), biometric data,

GPS localization, bank information and private messages. Non-sensitive data is any dynamic information that do not identify the user, often, this kind of information if public or shared, such as application source code.

Every program computes on either sensitive data or non-sensitive data, and this data follows a specific flow, first it is acquired from data sources and sent to data sinks (MCCABE, 2003). Data sources, in the context of mobile and IoT devices, are defined as method calls that read data from shared resources such as phone calls, screenshots, sensor polling data from ambient, device identification numbers (RASTHOFER; ARZT; BODDEN, 2014). Data sinks are methods calls that have at least one argument, this argument is non-constant data from the source code (RASTHOFER; ARZT; BODDEN, 2014). The data sink can be an interface to the user or system API for communication to other devices or store data (VIET TRIEM TONG; CLARK; MÉ, 2010).

To address the data privacy issue, mobile operating systems such as Android and IOs have a permission based system intended to control which sensitive information the application have access, like contact numbers, photos, videos and data from camera and microphone. These permissions are granted by the user at the installation or during the application usage (REQUEST APP PERMISSIONS. GOOGLE ANDROID DOCUMENTATION. (2018)). An issue existing in the permission system is called superprivileges, where the apps asks for unnecessary permissions, caused by developers that misunderstanding the API (FELT et al. (2011)). There is a methodology that help the developers to have a broader knowledge of the permissions in Android that has been proposed by BARRERA et al. (2010).

Another issue created when using the permission system is the abuse of privilege given by the user to an app, or even, the application can intentionally send information to a malicious host. As JIANG; ZHOU (2012) reports, there are malwares that are capable to steal sensitive information from the user, like SMS. To solve unwanted sensitive information flow inside Android applications, ARZT et al. (2014), WEI et al. (2014) and GORDON et al. (2015) developed static and dynamic ways to enforce that this kind of information does not leak to unwanted sink methods. The dynamic way to enforce information flow is called Dyanmic Flow Enforcement and statically is called Static Flow Enforcement. These methods are going to be presented in a deeper fashion in Chapter 2.

## 1.2 Objective

This work has two main objectives: generate a public database containing methods from Android APIs and evaluate the performance of the state of the art classification algorithms on this database. The methods should be classified into source or sink of sensitive information, or neithernor.

The motivation behind the database creation comes when RASTHOFER; ARZT; BODDEN (2014) developed the initial classification work. Their objective were create a machine-

learning solution to identify sources and sink methods from the code of any Android API. So, every time that you wanted to test other classification algorithms, you had to extract the methods and features if you evaluate in other API. This implies in extracting the API from a real Android device, which can be tricky if the person does not have this specific knowledge. Then, create a public database extract knowledge and develop better solutions to this problem is very likely.

The objective of evaluate different classification algorithms comes as RASTHOFER; ARZT; BODDEN (2014) shows results for only three classification algorithms, SVM, Decision Tree and Naive Bayes. From that, emerged the question if other classification algorithms have better results in this database. So, the second objective is to evaluate the most used classifiers in the literature, SVM, Naive Bayes, Decision Trees, MLP, KNN, including ensemble classifications methods.

## 1.3 Work Structure

This work is divided as follows: Chapter 2 is a short introduction about Static and Dynamic Analysis, followed by a explanation about Flow Enforcement techniques, which is important to understand what issues these methods addresses and what are the limitations, we also introduce the classification algorithms used and explain why we selected them. In Chapter 3 we introduce the methodology used in this work and the training and test architectures used by the machine learning methods. Chapter 4 we show the results archived, the dataset created and the scores of each classifier used in this work and we report our conclusions in Chapter 5.

# 2

# Background

In this chapter, we introduce in Section 2.1 the techniques used in flow enforcement and the issues of having a poor list of source and sink methods to make this kind of analysis. Also in this chapter, in Section 2.2 we present the classification algorithms used in this work and briefly explain why we selected them.

## 2.1 Analysis Techniques

There are two ways of making a security analysis, using Static Analysis and Dynamic Analysis. Static Analysis use the application source code to extract all paths of execution and infer if there exists some kind of malicious code. Dynamic Analysis needs to execute the application to and uses the execution path in many application runs (TAM et al., 2017). Static and Dynamic Analysis are used to enforce information flows using rules called flow policies (FERNANDES et al., 2016). These flow policies can be enforced during software execution or can be used to create a issue report to developer during compilation, called Dynamic Flow Enforcement and Static Flow Enforcement respectively. Dynamic Enforcement comes with a bigger overhead to the application as the flow enforcement is done during execution.

Some drawbacks can be seen when comparing static or dynamic analysis, the capability of detecting implicit flows and the quantity of false positives as result of these analysis. There are two ways of a information flow in the program: explicit and implicit, a explicit flow is a direct assignment of information to another variable or passing this information when calling a method. On the other hand, implicit flows are obfuscated data assignments by some control instruction like a conditional statement or loop. As the static analysis evaluates the whole source code, implicit flows can easily be detected, in contrast, the dynamic analysis can not detect that this flows exists, or, is needed to execute the code dozen of times to track all flows in the code (RUSSO; SABELFELD, 2010). In Figure 2.2, we can see an explicit flow of sensitive information and later, the sensitive information will flow to a sink method. In Figure 2.1 we can observe also an explicit flow when assigning the sensitive information to the variable and a implicit flow, when the sensitive information flow to the sink method.

Dynamic Flow Enforcement are techniques that tracks and enforces information flow

```
int sensitive_data = SensitiveSource();
int other_data = NonSensitiveSource();

NotSinkNorSource(other_data);
Sink(sensitive_data);
```

**Figure 2.1:** Explicit information flow in the variable assignment and sink method call.

```
int sensitive_data = SensitiveSource();
int other_data = NonSensitiveSource();

if(condition)
   Sink(sensitive_data);

NotSinkNorSource(other_data);
```

**Figure 2.2:** Explicit information flow in the variable assignment and implicit information flow in method call.

during the application runtime. These methods relies on Taint Analysis to track possible sensitive data flow to untrusted sinks. Taint Analysis marks every sensitive data gathered from a source and every other variable that inherit any operation from the tainted data, in the end, if any tainted variable is accessed by a sink method, the information has leaked and the analysis gives a detailed path through which the data passed. During the tracking, there are different methods to enforce in runtime that the data will not leak, FERNANDES et al. (2016) uses virtualization to guarantee that the data will only operate in the controlled environment and SUN et al. (2017) declassifying information before it is computed in trusted methods or if reach a trusted API.

Static Flow Enforcement starts by creating abstract models of the application code to provide a simpler representation (MYERS, 1999), using frameworks like Soot (VALLÉE-RAI et al., 2000). Then, this model will be used in control-flow, data-flow and points-to analysis to observe the application control, data sequence and compute static abstractions for variables (LI et al., 2017). These methods are implemented and used in DroidSafe GORDON et al. (2015). JFlow MYERS (1999) inserts statically checked and secured code when the application computes on sensitive data. There are also Static Enforcement techniques that uses taint analysis, like ARZT et al. (2014) that created a precise method of static information flow tracking using taint analysis.

Both Static and Dynamic Flow Enforcement techniques requires information about which methods are sources of sensitive data and which are data sinks. This is used to identify if a sink method is truly leaking sensitive data or not, for example, a sink method can send a variable

that does not have sensitive information during the method call, but during other execution, the variable can be storing sensitive data. This can be seen in two different ways when using static and dynamic analysis. So, lists containing sources and sinks of sensitive data are hand created, but this solution is impractical considering a huge API like the Android API (RASTHOFER; ARZT; BODDEN, 2014).

Other major issue of having a poorly created list of methods is the possibility of not tracking sensitive information, as the way to track sensitive information is based in taint every variable that operates in this kind of information, not having a complete list of sensitive data sources. In other hand, an excessive quantity of false sensitive sources and sinks can lead to a big quantity of false positives data leakage.

Considering that issue, RASTHOFER; ARZT; BODDEN (2014) proposed to use machine learning to automatically create a categorized list of sources and sinks methods to be used in Flow Enforcement techniques. The list consists in methods classified into Flow Classes and Android Method Categories. The Flow Classes are source of sensitive data, or just source, and sink of data, but also, the method can be neither source or sink. For Android Methods Categories, there are 12 different classes: account, Bluetooth, browser, calendar, contact, database, file, network, NFC, settings, sync, a unique identifier, and no category if the method does not belong to any of the previous.

The authors shortly compared Decision Tree and Naive Bayes with the SVM and chose to use SVM to create the categorized list of sources and sinks, as SVM showed to be more precise in categorize the Android methods. To classify, the authors utilize features extracted from the methods, like the method name, if the method has parameters, the return value type, parameter type, if the parameter is an interface, method modifiers, class modifiers, class name, if the method returns a value from another source method, if one parameter flows into a sink method, if a method parameters flows into a abstract sink and the method required permission. To categorize the methods, were used features like class name, method invocation, body contents, parameter type and return value type. After that, the methods list is generated containing if it is a sink, source and the method category.

## 2.2 Classification Methods

The classification approach have already showed better results than using the list of hand classified methods, but comparing only three algorithms can be a limiting factor. In this work we used Monolithic classifiers and Multi Classifier Systems (MCS). Monolithic classifiers are single classifiers commonly known, in this work we used Decision Tree, Naive Bayes, K-Nearest-Neighbors (KNN), Support Vector Machine (SVM), Multi Layer Perceptron (MLP), and Random Forest. The MCS classifiers, uses a combination of Monolithic classifiers or selects the best classifier, in a set of different classifiers of a same class, to be used in a determined sample space (CRUZ; SABOURIN; CAVALCANTI, 2018). The MCS algorithms used in this

work are KNORA-E, KNORA-U, META-DES, OLA, Single Best and Static selection, using Decision Tree and Perceptron as main classifiers.

Following the methodology presented by RASTHOFER; ARZT; BODDEN (2014), we selected the best classifier reported, SVM with linear kernel and the other classifiers evaluated, Naive Bayes and Decision Tree. To have a complete comparison for the Monolithic classifiers we added KNN, MLP and Random Forest to the evaluation.

A straight question when using MLP is if Deep Learning methods can outperform these monolithic classifiers by a signifcant margin. As stated by LECUN; BENGIO; HINTON (2015), the objective of Deep Learning is to create intermediate representations of raw data, like pixels in a image, to ease the classification problem, for example in image recognition and speech recognition. KONTSCHIEDER et al. (2015) use this premise to use a Deep Convolution Neural Networks to create features for a Decision Tree, having better results than the GoogleLeNet (SZEGEDY et al., 2015) in the problem of image classification and detection. Thus the usage of this technique is more suitable if the the dataset created does not have lexical features which would be learned in the process. YANG; MORILLO; HOSPEDALES (2018) evaluated Deep Neural Decision Trees in datasets that do not benefits from creation of intermediate representations, reporting that the method does not outperformed standard Decision Trees.

There are some definitions we must introduce before explain the choosing method for MCS algorithms. First is the definition of Dynamic Classifier Seleciton (DCS) and Dynamic Ensemble Selection (DES). DCS methods choose only the best from a pool of classifiers, this selection is done by using a region of competence to evaluate the accuracy. Otherwise, DES techniques selects more than one classifier to make the prediction and can combine them using methods of Fusion of Label methods, like Majority Vote, Weighted Majority Vote, Naive Bayes, Multinomial Methods, Probabilistic Approximations and Singular Decomposition (KUNCHEVA, 2004).

Intuitively, the MCS should have a better result when comparing to Monolithic classifiers, as the objective is to use a combination of many Monolithic classifiers, MCS should have better results (KUNCHEVA, 2004). CRUZ; SABOURIN; CAVALCANTI (2018) made a review about the performance of MCS algorithms over 30 different datasets, so we selected the best classifiers of each class to perform the evaluation on our dataset. OLA have a DCS selection approach using accuracy as criteria, KNORA-E and KNORA-U are oracle based DES algorithms, where KNORA-E selects all classifiers selected are the ones which have no wrong classifcations in the region of competence, and KNORA-U uses all classifiers that had at least one right classification in the region of competence, the META-DES is a meta-learning classifier where uses a classifier to select the best classifier in a region of competence. Also we used the baseline methods, Single Best and Static Selection to perform the evaluation. With these algorithms, we must have a better understanding about the performance of MCS in this classification problem.

# 3

# Architecture

The proposed architecture consists in two steps, a training step and test step: the training step is shown in Figure 3.1 and the test tep is shown in Figure 3.2. The training step consists in extract features from real Android methods and use that combined with the real classes to train classifiers. The first step to create the classifiers to is to have a list of fully implemented Android APIs and a list of already classified methods.

The APIs used in this work have been selected from two different repositories: the default repository of Android images used by RASTHOFER; ARZT; BODDEN (2014) and the second repository is a well known set of Android images, provided by ANDROID HIDDEN APIS (2018), which is used by mobile developers that need methods available only in fully implemented APIs, that are only available when the API is extracted from real devices. The APIs from level 3 to 17 used in this work were extracted from SUSI ANDROID APIS (2018) and the APIs 17, 19 and 21 to 27 gathered from ANDROID HIDDEN APIS (2018). These two APIs repositories meet the requirement imposed by RASTHOFER; ARZT; BODDEN (2014), and consists that the APIs used need to have the fully method implementation, done by extracting APIs from real devices.

If the extraction is made using an emulator, instead of a real device, the API comes without the complete method implementation, called method stubs. As long as the taint tracker try to execute or extract runtime information from any of these methods stubs, an exception is thrown and the execution is stopped. The fully implemented methods are important due to information tracking that is done during the feature extraction, if a API method calls another method that is a source of sensitive data, the method potentially be a source of data too. Another point is that some of syntactic features are extracted from data flow inside those methods, if no method implementation were given, those features could not be computed.

In addition to the APIs, it is necessary to have a starting list of Android methods that have been hand classified, called hand annotated methods. The feature extractor will use the hand annotated methods and the API to generate the methods features and if possible, infer the classes of non annotated methods using static taint analysis. The result of this step is the dataset used in the classification and is described in more details in Section 4.1. With the hand annotated methods and the APIs in hand, the following step is to apply the feature extractor to each API,

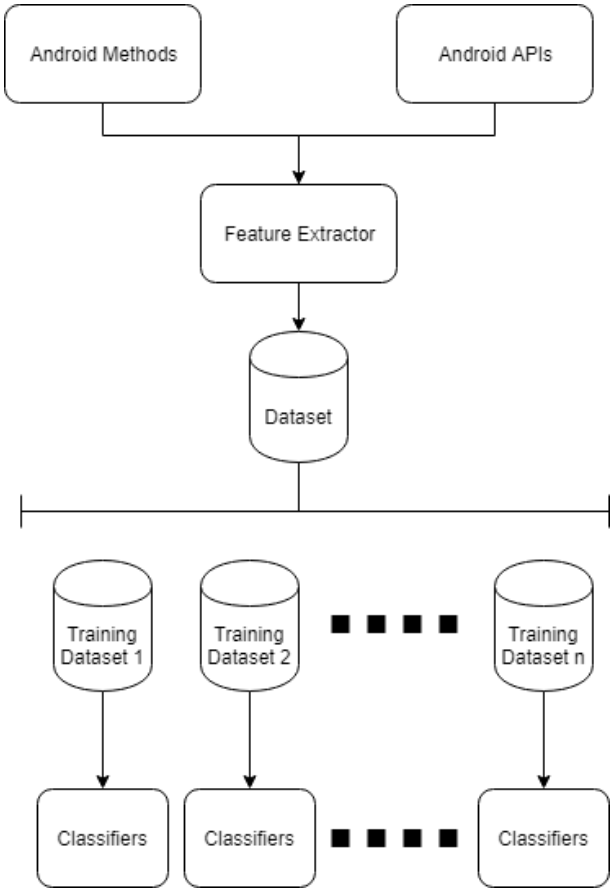


**Figure 3.1:** Overview of the proposed training flow. First, the feature extractor uses Android Methods and Android APIs to create a dataset, which will be sampled into other 30 different datasets that will be used to train the classifiers.

concatenate all the results and remove all the repeated entries, resulting in the dataset.

After the dataset creation, 30 other datasets are randomly sampled from the original, with the objective to train and evaluate the classification algorithms using a hypothesis test. Each of these datasets are subdivided into two disjoint datasets: train and test. The train dataset consists in 80% of the original dataset and to test the model effectiveness, the test dataset have 20% of the original samples. This is a random division done in such a way that the resultant datasets maintain the classes proportion observed in the original, if the original has 45% of source methods, the train and test must have a proportion close to that. The datasets are created by random sample of the original one, the algorithm calculates the classes proportion and randomly selects entries to be used in train and test keeping the proportion close to the original one.

At the end of one training step, the model is evaluated with the test dataset. The whole test step is shown in Figure 3.2, for each dataset, there is a classifier to categorize the samples. This will result in a list of samples and classified labels that are going to be compared with the original label. This generates the evaluation results, consisting in precision, recall, F1 score and accuracy, the result of each metric is saved. After the the result saved, the model is overwritten

and a fresh one is used to be used back in the training step. This ensures that no information is being transferred to the next step, forgetting all the information previously learned. And at the end of 30 steps of training and test, the mean and standard deviation of each metric is calculated and reported in the results, Section 4.4.

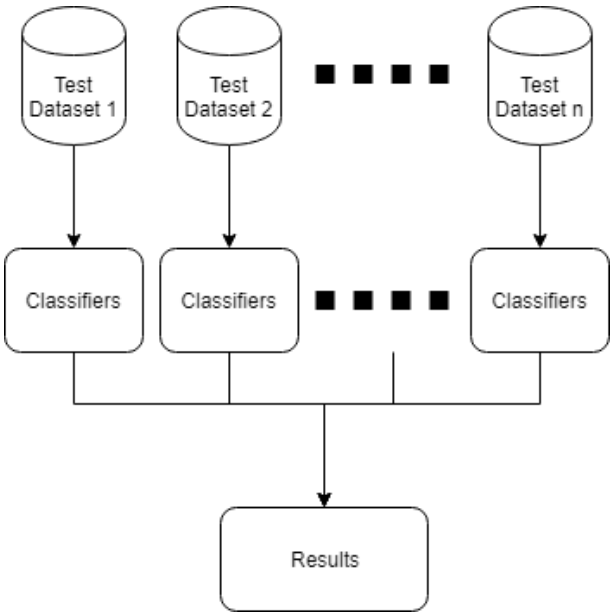


**Figure 3.2:** Overview of the proposed testing flow. After the creation of datasets, the result of each classifier is used in evaluation.

# 4

## Results

In this chapter, the results are exposed in the following way: Section 4.1 shows how the dataset has been created and the overall proportion of classes. Section 4.2 is reserved to comparison and evaluation of monolithic machine learning models, Decision Tree, K-Nearest Neighbors, Multi Layer Perceptron, Naive Bayes and Support Vector Machine. For Multiple Classifier System (MCS), the Section 4.3 compares the KNORA-E, KNORA-U, META-DES, OLA, Single Best and Static Selection. After the comparison between Monolithic and MCS, the overall comparison is discussed in Section 4.4.

Using the dataset described in Section 4.1 other 30 different datasets have been randomly sampled that are subdivided into two: 80% of the original dataset for model training and 20% to test the model effectiveness. This method intends to make a Hypothesis Test, this help to statically evaluate if a classifier is really effective when applied in this dataset. After the classifiers applied to each of the 30 datasets, the mean and standard deviation of each metric are calculated and displayed in the result tables. For each classification method, both Monolithic and MCS have its setup described in Sections 4.2 and 4.3 to ease future replications of this work.

Each classifier is evaluated with 4 different metrics, precision (equation 4.1), recall (equation 4.2), F1 score (equation 4.3) and accuracy (equation 4.4), the classifier with bigger values in these metrics have the best result, they vary from 0 to 1. The precision is the ratio of correctly predictions to the total predictions done. Accuracy is the ratio of correctly true predictions to the total of predictions. Recall is the ratio of correctly positive predictions to all the predictions of a class. F1 score represents the harmonic mean between precision and recall, which is a more meaningful metric than the mean between precision and recall SASAKI et al. (2007). True positives are all instances of a class $C$ that are correctly classified. True negatives are all instances that do not belong to $C$ and are correctly classified. False positives of a class $C$ are the instances of other classes that has been classified as $C$. False negatives are instances of $C$ that has been classified as not belonging to $C$.

$$precision = \frac{TP}{TP+FP} \quad (4.1) \qquad F1 = \frac{2 \times recall \times precision}{recall + precision} \quad (4.3)$$

$$recall = \frac{TP}{TP+FN} \quad (4.2) \qquad accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (4.4)$$

## 4.1 Dataset

The dataset created in this work uses the feature extractor developed by RASTHOFER; ARZT; BODDEN (2014). The extractor creates meaningful information using the Android methods names and their real implementation in the Android API. As result, the extractor gives the method class, if has been hand annotated, and a list of features. These features are lexical, semantic and syntactic, which contains information about the method name, parameters, return type, method modifiers, classes modifiers, if exists data flow in the method return or parameters and the required permissions.

Lexical features follows the idea that the Android APIs adopts a specific coding style, described in AOSP JAVA CODE STYLE FOR CONTRIBUTORS (2018). Extracting if a method name or parameter name contains certain strings can lead to the prediction of the method class. For syntactic features, the feature extractor evaluates if exists data flow inside a method using Taint Analysis, discussed in Chapter 2. Finally, the semantic features are information extracted from classes modifiers, such as private, public, protected and types of variables, arguments and methods returns.

As result, the feature extractor gives 215 features, consisting in 53 semantic, 45 syntactic and 117 lexical features, extracted from the Android API Level 3 to Level 27, excluding APIs Level 18 and 20 which complete binaries could not be found. The features extracted consists in boolean variables represented in 12 categories, the dataset also contains the full method name and signature but is not being used in classification. The lexical features are extracted by analyzing the stream of characters from the source code, syntactic features represents the dependency of data and control for code variables and methods, semantic features consists in the types and modifiers for variables, methods and classes AHO (2003).

| Source | Sink | Neithernor | Total |
|---|---|---|---|
| 375 (55.97 %) | 176 (26.27 %) | 119 (17.76 %) | 670 (100 %) |

**Table 4.1:** The final dataset proportion without duplicated entries.

With that said, Table 4.1 shows the proportion of each class for the dataset. First we evaluate the feature extractor using the Android API 17, same API used in RASTHOFER; ARZT; BODDEN (2014) and we could extract a total of 535 methods, consisting in 131 sinks, 88 sources and 316 neither nor. Using Android API Level 3 to Level 27, excluding APIs Level 18 and 20, we extracted 670 methods in total when dropping repeated entries (Dropped Entries dataset), being 176 sinks, 119 sources and 375 neither nor. It is important to observe that none of the dataset entries are duplicated, but if we look closely to the dataset, the same method can have different value of features through different APIs, as shown in Figure 4.1. So, considering method names as duplication factor (Dropped Names dataset), we end up with only 543 method, unlike the

670 for the Dropped Entries, disperse in 134 sinks, 87 sources and 322 neithernor. As different feature values consists in different entries, despite the same method name, we considered the larger dataset in order to have a bigger quantity of data to be analyzed, so we only dropped entries that are really duplicated. Sometimes the feature extractor could not infer syntactic features for a method, these entries are also removed from the dataset to keep the dataset integrity.

Due to the repetition of methods in different APIs, and not necessarily the same features repeated, drop entries with repeated method names reduces the quantity of methods available to be trained. So we chose as default dataset for the evaluation the dataset with dropped repeated entries, this dataset have more information to be learned and already guarantee the division in training and testing to be disjoint as have no repetition.

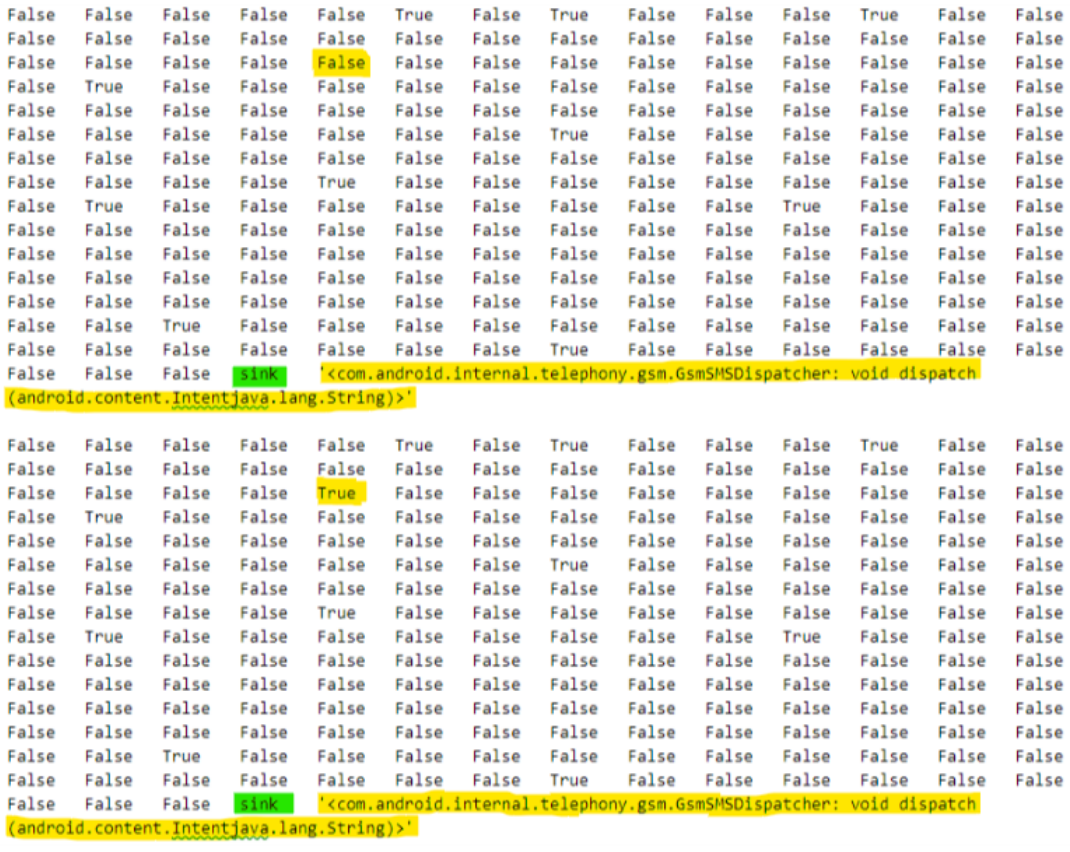


**Figure 4.1:** Methods through different API Levels. We can observe in yellow the feature "Method modifier is PUBLIC" and the full method name. Even for different features values, marked in green, the method still belongs to the sink class.

## 4.2 Monolithic

For the Monolithic classifiers, we have the Decision Tree, Multinomial and Bernoulli Naive Bayes, K-Nearest Neighbor (KNN), Support Vector Machine (SVM), Multi-Layer Perceptron and Random Forest.

The setup for Decision Tree is the split strategy to select the most relevant feature with Gini Impurity as criterion. In Multinomial and Bernoulli Naive Bayes we choose a $\alpha$ of 1.

For KNN, we use 5 as the number of neighbors and a uniform weight for the neighbor points. The SVM has L2 as penalty, linear kernel, $C = 1$ and tolerance of $10^{-4}$. For MLP we used L2 regularization with alpha of $10^{-4}$, 1000 neurons in hidden layer, no early stopping, activation layer as relu, max of 1000 iterations and tolerance of $10^{-4}$. For Random Forests we are using 10 estimators, the minimum samples to split is 2 and Gini Impurity as criterion. The parameters can also be seen in Table 4.2, these parameters are the default ones, in this work we are not using optimization.

| Model | Parameters | | | | |
|---|---|---|---|---|---|
| Decision Tree | criterion Gini | spliter best | | | |
| Multinomial NB | $\alpha = 1$ | | | | |
| Bernoulli NB | $\alpha = 1$ | | | | |
| KNN | $k = 5$ | uniform weight | | | |
| SVM | linear kernel | penalty L2 | $C = 1$ | tolerance $10^{-4}$ | |
| MLP | $\alpha = 10^{-4}$ | reg. L2 | activation relu | tolerance $10^{-4}$ | hidden neurons 1000 |
| Random Forest | criterion Gini | samples 2 | estimators 10 | | |

**Table 4.2:** Parameters for Monolithic Classifiers

Table 4.3 is showing the results for Monolithic Classifiers. We can observe that the MLP has better results overall in all metrics evaluated and the difference between MLP, Random Forest and SVM does not surpass 0.1. These results are within the confidence interval, suggesting that a the Random Forest and a LinearSVM can solve the problem as good as a MLP.

## 4.3 Multiple Classifier System

For the Multiple Classifier System we used KNORA-E, KNORA-U, META-DES, OLA, Single Best and Static Selection. Each of these algorithms has the objective to select the best classifier in a set, in our evaluation, we are using a pool of 100 classifiers of the same base class.

The base classifier classes used are Perceptron and Decision Tree, in all ensemble algo-

| Model | Precision | Recall | F1 Score | Accuracy |
|---|---|---|---|---|
| Decision Tree | 0.8324 (0.0552) | 0.8318 (0.0625) | 0.8301 (0.0446) | 0.8413 (0.0281) |
| Multinomial NB | 0.8019 (0.0540) | 0.8022 (0.0632) | 0.7998 (0.0425) | 0.8204 (0.0282) |
| Bernoulli NB | 0.8075 (0.0549) | 0.7979 (0.0662) | 0.8003 (0.0456) | 0.8219 (0.0310) |
| KNN | 0.8560 (0.0488) | 0.7969 (0.0604) | 0.8175 (0.0448) | 0.8393 (0.0242) |
| SVM | 0.8790 (0.0480) | 0.8702 (0.0541) | 0.8728 (0.0392) | 0.8796 (0.0274) |
| MLP | **0.8838 (0.0474)** | **0.8709 (0.0544)** | **0.8758 (0.0426)** | **0.8856 (0.0287)** |
| Random Forest | 0.8821 (0.0510) | 0.8459 (0.0686) | 0.8586 (0.0469) | 0.8719 (0.0319) |

**Table 4.3:** Result for Monolithic Classifiers, Mean (Standard Deviation), the best classifier for each metric is highlighted in bold.

rithms we use the same base classifier configuration. For Perceptron we used no regularization, no early stopping, max of 1000 iterations and tolerance of $10^{-1}$. For Decision Tree, the split strategy is to select the most relevant feature with Gini Impurity as criterion. For ensemble algorithms, the KNORA-E and KNORA-U parameters were KNN to estimate the classifier competence using 7 neighbors, with no dynamic pruning and no indecision region. For META-DES, we are using Multinomial Naive Bayes as meta-classifier, 5 output profiles to estimate the competence using a KNN with 7 neighbors to decide the region of competence. And finally, static selection, we are choosing 50% of the base classifiers. All parameters can be seen in Table 4.4.

In Table 4.5 is presented the MCS algorithms using Decision Tree classifier. We can observe a better mean result using META-DES in all metrics, except when comparing the Accuracy with KNORA-E. Considering the mean and standard deviation, the results for KNORA-E, KNORA-U, OLA, Static Selection and META-DES are very close to the same interval. But we can observe that there is a bigger gap between OLA and Static Selection, and KNORA-E, KNORA-U and META-DES in Precision, Recall and F1. Comparing the Accuracy there is a smaller gap.

When using Perceptron as main classifier, Table 4.6, we can observe that KNORA-E have the best mean results in Recall, F1 and Accuracy but META-DES has a better Precision. When considering mean and standard deviation, the results of KNORA-E, KNORA-U, META-DES, Static Selection and OLA are in the same interval for all the metrics. The difference between KNORA-E and META-DES is smaller in Precision, F1 and Accuracy but has a smaller Recall comparing to OLA.

| Model | Parameters | | |
|---|---|---|---|
| Perceptron | max iterations 1000 | tolerance $10^{-4}$ | |
| Decision Tree | criterion Gini | spliter best | |
| KNORA-E and KNORA-U | $k = 7$ | no prune | no indecision |
| META-DES | Multinom. NB | $K_p = 5$ | $k = 7$ |
| Static Selection | selection 50% | | |

**Table 4.4:** Parameters for Multiple Classifier System

| Model | Precision | Recall | F1 Score | Accuracy |
|---|---|---|---|---|
| KNORA-E | 0.8602 (0.0517) | 0.8395 (0.0656) | 0.8469 (0.0455) | **0.8580 (0.0306)** |
| KNORA-U | 0.8377 (0.0557) | 0.8071 (0.0743) | 0.8181 (0.0495) | 0.8313 (0.0321) |
| META-DES | **0.8609 (0.0551)** | **0.8423 (0.0639)** | **0.8492 (0.0482)** | 0.8575 (0.0344) |
| OLA | 0.8263 (0.0589) | 0.8104 (0.0667) | 0.8158 (0.0532) | 0.8306 (0.0387) |
| Single Best | 0.7609 (0.0739) | 0.7356 (0.0994) | 0.7423 (0.0688) | 0.7629 (0.0447) |
| Static Selection | 0.8424 (0.0579) | 0.8099 (0.0741) | 0.8213 (0.0492) | 0.8338 (0.0341) |

**Table 4.5:** Result for Multiple Classifier System, Mean (Standard Deviation), using Decision Tree as main classifier with most relevant feature as split strategy and Gini Impurity as criterion. The best classifier for each metric is highlighted in bold

The results for best classifier using Decision Tree has close results to the best classifier using Perceptron. Comparing KNORA-E with Perceptron and META-DES with Decision Tree, the META-DES with Decision Tree has better results in all metrics, but again the difference between both is so small that do not exceed the confidence interval.

## 4.4 Final Considerations

With the Monolithic and MCS compared, now we must compare the best Monolithic and the best MCS to fully understand which model is the most indicated to solve the problem of

| Model | Precision | Recall | F1 Score | Accuracy |
|---|---|---|---|---|
| KNORA-E | 0.8607 (0.0556) | **0.8368 (0.0678)** | **0.8452 (0.0471)** | **0.8570 (0.0273)** |
| KNORA-U | 0.8446 (0.0562) | 0.8148 (0.0581) | 0.8260 (0.0451) | 0.8386 (0.0301) |
| META-DES | **0.8748 (0.0559)** | 0.8167 (0.0700) | 0.8375 (0.0518) | 0.8525 (0.0335) |
| OLA | 0.8418 (0.0580) | 0.8216 (0.0550) | 0.8282 (0.0413) | 0.8410 (0.0268) |
| Single Best | 0.7913 (0.0753) | 0.7552 (0.1007) | 0.7644 (0.0688) | 0.7866 (0.0451) |
| Static Selection | 0.8523 (0.0593) | 0.8158 (0.0717) | 0.8291 (0.0512) | 0.8438 (0.0329) |

**Table 4.6:** Result for Multiple Classifier System, Mean (Standard Deviation), using Perceptron as main classifier using no regularization, no early stopping, max of 1000 iterations and tolerance of $10^{-1}$. The best classifier for each metric is highlighted in bold

Android API methods categorization. Table 4.7 show the performance of each classifier: MLP and METADES using Decision Tree, and we can see that the MLP has better results in all the metrics.

| Model | Precision | Recall | Accuracy | F1 Score |
|---|---|---|---|---|
| MLP | **0.8838 (0.0474)** | **0.8709 (0.0544)** | **0.8758 (0.0426)** | **0.8856 (0.0287)** |
| META-DES Decision Tree | 0.8609 (0.0551) | 0.8423 (0.0639) | 0.8492 (0.0482) | 0.8575 (0.0344) |

**Table 4.7:** The best of Monolithic compared to the best of MCS, Mean (Standard Deviation). The best classifier for each metric is highlighted in bold.

Looking deep to the results for each class, we can observe in Table 4.8 that the MLP has a better score in most classes and metrics whereas METADES using Decision Tree as base classifier has better Precision when classifying the Source methods. Other point to be considered is the standard deviation, in this case we can conclude that, as the classifiers are also in the same interval for all the metrics, this is another evidence that a simpler algorithm can address this problem.

To understand if the models are statistically equal, we used the Wilcoxon Sign-test. This test compares two samples comes from the same distribution, the hypothesis is defined as the samples comes from different distributions. If this hypothesis is rejected, $p > \alpha$ then they come from the same distribution, if is not rejected, then they come from different distributions. This is a nonparametric test, implying that we can use to discrete or non-normal distributions. In Table

| Model | Class | Precision | Recall | Accuracy | F1 Score |
|---|---|---|---|---|---|
| MLP | (1) | 0.8792 (0.0597) | **0.8708 (0.0786)** | **0.8731 (0.0577)** | **0.8856 (0.0287)** |
| | (2) | **0.8803 (0.0486)** | **0.8219 (0.0598)** | **0.8489 (0.0450)** | **0.8856 (0.0287)** |
| | (3) | **0.8920 (0.0339)** | **0.9200 (0.0248)** | **0.9055 (0.0251)** | **0.8856 (0.0287)** |
| METADES Decision Tree | (1) | **0.8870 (0.0658)** | 0.8611 (0.0889) | 0.8702 (0.0580) | 0.8575 (0.0344) |
| | (2) | 0.8344 (0.0636) | 0.7676 (0.0691) | 0.7983 (0.0579) | 0.8575 (0.0344) |
| | (3) | 0.8614 (0.0361) | 0.8982 (0.0338) | 0.8790 (0.0288) | 0.8575 (0.0344) |

**Table 4.8:** The best of Monolithic compared to the best of MCS, Mean (Standard Deviation). The Class (1) represents the Source methods, Class (2) are the Sink methods and (3) Neithernor. The best classifier for each metric and class is highlighted in bold.

4.9 we used the Wilcoxon Sign-test to evaluate if the best algorithms for Monolithic and MCS are pairwise equal. Then we can observe that the only classifiers that are not pairwise equal are MLP compared with META-DES using Decsion Tree, KNORA-E using Decision Tree compared with META-DES using Decsion Tree, META-DES using Decsion Tree compared with KNORA-E using Perceptron and KNORA-E using Decision Tree and KNORA-E using Perceptron. All the other pairwise combinations between MLP, SVM, Random Forest, KNORA-E and KNORA-U with Decision Tree or Perceptron and META-DES with Decision Tree or Perceptron comes from the same distribution with at least a significance level of 0.01.

| Hypothesis | $p$ |
|---|---|
| SVM vs META-DES Perceptron | 0.000 ●●● |
| MLP vs META-DES Perceptron | 4.33e-15 ●●● |
| SVM vs Random Forest | 3.05e-12 ●●● |
| META-DES Decision Tree vs META-DES Perceptron | 1.57e-11 ●●● |
| KNORA-E Perceptron vs META-DES Perceptron | 2.60e-08 ●●● |
| MLP vs Random Forest | 4.56e-08 ●●● |
| KNORA-E Decision Tree vs META-DES Perceptron | 4.60e-07 ●●● |
| SVM vs KNORA-E Perceptron | 1.95e-05 ●●● |
| Random Forest vs META-DES Decision | 2.45e-05 ●●● |
| SVM vs KNORA-E Decision Tree | 1.29e-04 ●●● |
| SVM vs META-DES Decision Tree | 0.0027 ●●● |
| Random Forest vs META-DES Perceptron | 0.0047 ●●● |
| Random Forest vs KNORA-E Perceptron | 0.0077 ●●● |
| Random Forest vs KNORA-E Decision Tree | 0.0083 ●●● |
| MLP vs KNORA-E Perceptron | 0.0109 ●● |
| MLP vs KNORA-E Decision | 0.0184 ●● |
| SVM vs MLP | 0.0425 ●● |
| MLP vs META-DES Decision Tree | 0.1380 |
| KNORA-E Decision Tree vs META-DES Decision Tree | 0.2498 |
| META-DES Decision Tree vs KNORA-E Perceptron | 0.3088 |
| KNORA-E Decision Tree vs KNORA-E Perceptron | 0.9430 |

**Table 4.9:** Pairwise comparison using Wilcoxon Sign-test, the hypothesis are ordered in ascending order according to the $p$-value, rejected hypothesis have $\alpha = \{0.1, 0.05, 0.01\}$ and are marked with ●, ●●, ●●● respectively.

# 5

# Conclusion

The information leakage is a big concern of mobile users and developers, there are many security systems and tools that uses taint tracking to identify unwanted information flow into a IoT and mobile application. These tools need a precise list of sources and sinks methods to detect malicious flows. This is solved using classification algorithms to generate this list, focusing in maximize the quantity of true positives and false negatives leakage.

The problem of categorize Android API methods come along with some difficulties: the lack of substantiated databases, lack of precise public information about the methods in APIs and the big variety of APIs available to Android. This work provides a database to develop classification strategies and increase the performance of information tracking systems.

We can conclude that the MLP is the best classifier to categorize Android methods, based on overall and per class performance, it has scored better results in almost all metrics by class and overall. Also, that other classifiers can be used as alternative to the MLP, such as META-DES with Decision Tree and SVM. These classification algorithms are at the same baseline value, the precision difference between MLP and META-DES using Decision Tree is 0.011 and recall of just 0.0089, the statistical testing also points that these algorithms are not equivalent but SVM and META-DES with Decision Tree are equivalents.

References

AHO, A. V. **Compilers**: principles, techniques and tools (for anna university), 2/e. Pearson Education India, 2003.

AL-SARAWI, S. et al. Internet of Things (IoT) communication protocols. In: INFORMATION TECHNOLOGY (ICIT), 2017 8TH INTERNATIONAL CONFERENCE ON. **Anais** 2017. p.685–690.

ANDROID Hidden APIs. Accessed: 01-11-2018. https://github.com/anggrayudi/android-hidden-api.

AOSP Java Code Style for Contributors. Accessed: 12-10-2018. https://source.android.com/setup/contribute/code-style.

ARZT, S. et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. **Acm Sigplan Notices**, v.49, n.6, p.259–269, 2014.

BARRERA, D. et al. A methodology for empirical analysis of permission-based security models and its application to android. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 17. **Proceedings** 2010. p.73–84.

CRUZ, R. M.; SABOURIN, R.; CAVALCANTI, G. D. Dynamic classifier selection: recent advances and perspectives. **Information Fusion**, v.41, p.195–216, 2018.

FELT, A. P. et al. Android permissions demystified. In: ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 18. **Proceedings** 2011. p.627–638.

FERNANDES, E. et al. FlowFence: practical data protection for emerging iot application frameworks. In: USENIX SECURITY SYMPOSIUM. **Anais** 2016. p.531–548.

GORDON, M. I. et al. Information Flow Analysis of Android Applications in DroidSafe. In: NDSS. **Anais** 2015. v.15, p.110.

GROWTHENABLER. **Market Pulse Report, Internet of Things**. Accessed: 05-03-2018. https://growthenabler.com/flipbook/pdf/IOT

JIANG, X.; ZHOU, Y. Dissecting android malware: characterization and evolution. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 2012. **Anais** 2012. p.95–109.

KOLIAS, C. et al. DDoS in the IoT: mirai and other botnets. **Computer**, v.50, n.7, p.80–84, 2017.

KONTSCHIEDER, P. et al. Deep neural decision forests. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER VISION. **Proceedings** 2015. p.1467–1475.

KUNCHEVA, L. I. **Combining pattern classifiers**: methods and algorithms. John Wiley & Sons, 2004.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **nature**, v.521, n.7553, p.436, 2015.

LI, L. et al. Static analysis of android apps: a systematic literature review. **Information and Software Technology**, v.88, p.67–95, 2017.

MCCABE, J. D. **Network Analysis, Architecture and Design, Second Edition (The Morgan Kaufmann Series in Networking)**. Morgan Kaufmann, 2003.

MINERVA, R.; BIRU, A.; ROTONDI, D. Towards a definition of the Internet of Things. **IEEE Internet Initiative**, v.1, p.1–86, 2015.

MYERS, A. C. JFlow: practical mostly-static information flow control. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 26. **Proceedings** 1999. p.228–241.

RAJ, P.; RAMAN, A. C. **The Internet of Things**: enabling technologies, platforms, and use cases. Auerbach Publications, 2017.

RASTHOFER, S.; ARZT, S.; BODDEN, E. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In: NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM NDSS. **Anais** 2014.

REQUEST App Permissions. Google Android Documentation. Accessed: 12-10-2018. https://developer.android.com/training/permissions/requesting.

RUSSO, A.; SABELFELD, A. Dynamic vs. static flow-sensitive security analysis. In: COMPUTER SECURITY FOUNDATIONS SYMPOSIUM (CSF), 2010 23RD IEEE. **Anais** 2010. p.186–199.

SASAKI, Y. et al. The truth of the F-measure. **Teach Tutor mater**, v.1, n.5, p.1–5, 2007.

SUN, C. et al. Data-Oriented Instrumentation against Information Leakages of Android Applications. In: IEEE 41ST ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 2017. **Anais** 2017. p.485–490.

SUSI Android APIs. Accessed: 01-11-2018. https://github.com/Sable/android-platforms.

SZEGEDY, C. et al. Going deeper with convolutions. In: IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION. **Proceedings** 2015. p.1–9.

TAM, K. et al. The evolution of android malware and android analysis techniques. **ACM Computing Surveys (CSUR)**, v.49, n.4, p.76, 2017.

VALLÉE-RAI, R. et al. Optimizing Java bytecode using the Soot framework: is it feasible? In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION. **Anais** 2000. p.18–34.

VIET TRIEM TONG, V.; CLARK, A. J.; MÉ, L. Specifying and enforcing a fine-grained information flow policy: model and experiments. **Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications**, v.1, n.1, p.56–71, 2010.

WEI, F. et al. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: ACM SIGSAC CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 2014. **Proceedings** 2014. p.1329–1341.

YANG, Y.; MORILLO, I. G.; HOSPEDALES, T. M. Deep Neural Decision Trees. **arXiv preprint arXiv:1806.06988**, 2018.

YOUNG, C. **Google's Newest Feature**: find my home. Accessed: 10-11-2018. https://www.tripwire.com/state-of-security/vert/googles-newest-feature-find-my-home/.

ZHANG, Z.-K. et al. IoT security: ongoing challenges and research opportunities. In: SERVICE-ORIENTED COMPUTING AND APPLICATIONS (SOCA), 2014 IEEE 7TH INTERNATIONAL CONFERENCE ON. **Anais** 2014. p.230–234.

# Appendix

**Table 1:** List of all features extracted. Class 1 means a lexical feature, Class 2 a semantic feature and 3 a syntactic feature.

| Class | Feature |
|---|---|
| 1 | Base name of class package name: accounts |
| 1 | Base name of class package name: io |
| 1 | Base name of class package name: music |
| 1 | Base name of class package name: telephony |
| 1 | Base name of class package name: webkit |
| 1 | Method has parameters |
| 1 | Method is lone getter or setter |
| 2 | Method is part of a ABSTRACT class |
| 2 | Method is part of a FINAL class |
| 2 | Method is part of a PRIVATE class |
| 2 | Method is part of a PROTECTED class |
| 2 | Method is part of a PUBLIC class |
| 2 | Method is part of a STATIC class |
| 2 | Method is part of an anonymous class |
| 2 | Method is part of class android.app.Activity |
| 2 | Method is part of class android.app.BroadcastReceiver |
| 2 | Method is part of class android.app.ContentProvider |
| 2 | Method is part of class android.app.Service |
| 2 | Method is part of class android.content.ContentResolver |
| 2 | Method is part of class android.content.Context |
| 2 | Method is part of class that contains the name com.google.common.io |
| 2 | Method is part of class that contains the name java.io. |
| 2 | Method is part of class that ends with Context |
| 2 | Method is part of class that ends with Factory |
| 2 | Method is part of class that ends with Handler |
| 2 | Method is part of class that ends with Loader |
| 2 | Method is part of class that ends with Manager |
| 2 | Method is part of class that ends with Service |
| 2 | Method is part of class that ends with View |
| 2 | Method is thread runner |
| 2 | Method modifier is FINAL |
| 3 | Method modifier is PROTECTED |
| 2 | Method modifier is PUBLIC |
| 2 | Method modifier is STATIC |
| 1 | Method name ends with Messenger |
| 1 | Method name starts with <init> |

| | |
|---|---|
| 1 | Method name starts with add |
| 1 | Method name starts with apply |
| 1 | Method name starts with bind |
| 1 | Method name starts with clear |
| 1 | Method name starts with close |
| 1 | Method name starts with delete |
| 1 | Method name starts with disable |
| 1 | Method name starts with dispatch |
| 1 | Method name starts with do |
| 1 | Method name starts with dump |
| 1 | Method name starts with enable |
| 1 | Method name starts with finish |
| 1 | Method name starts with get |
| 1 | Method name starts with handle |
| 1 | Method name starts with insert |
| 1 | Method name starts with is |
| 1 | Method name starts with load |
| 1 | Method name starts with note |
| 1 | Method name starts with notify |
| 1 | Method name starts with onClick |
| 1 | Method name starts with open |
| 1 | Method name starts with perform |
| 1 | Method name starts with process |
| 1 | Method name starts with put |
| 1 | Method name starts with query |
| 1 | Method name starts with register |
| 1 | Method name starts with release |
| 1 | Method name starts with remove |
| 1 | Method name starts with request |
| 1 | Method name starts with restore |
| 1 | Method name starts with run |
| 1 | Method name starts with send |
| 1 | Method name starts with set |
| 1 | Method name starts with start |
| 1 | Method name starts with supply |
| 1 | Method name starts with toggle |
| 1 | Method name starts with unregister |
| 1 | Method name starts with update |
| 2 | Method returns constant |

| | |
|---|---|
| 2 | Method starts with on and has void/bool return type |
| 2 | Parameter is interface |
| 3 | Parameter to abstract sink |
| 3 | Parameter to sink method adjust |
| 3 | Parameter to sink method bind |
| 3 | Parameter to sink method broadcast |
| 3 | Parameter to sink method clear |
| 3 | Parameter to sink method com.android.internal.telephony.CommandsInterface |
| 3 | Parameter to sink method connect |
| 3 | Parameter to sink method create |
| 3 | Parameter to sink method delete |
| 3 | Parameter to sink method dial |
| 3 | Parameter to sink method disable |
| 3 | Parameter to sink method dispatch |
| 3 | Parameter to sink method dump |
| 3 | Parameter to sink method enable |
| 3 | Parameter to sink method enqueue |
| 3 | Parameter to sink method insert |
| 3 | Parameter to sink method notify |
| 3 | Parameter to sink method onCreate |
| 3 | Parameter to sink method perform |
| 3 | Parameter to sink method println |
| 3 | Parameter to sink method put |
| 3 | Parameter to sink method remove |
| 3 | Parameter to sink method replace |
| 3 | Parameter to sink method restore |
| 3 | Parameter to sink method save |
| 3 | Parameter to sink method send |
| 3 | Parameter to sink method set |
| 3 | Parameter to sink method setup |
| 3 | Parameter to sink method show |
| 3 | Parameter to sink method start |
| 3 | Parameter to sink method sync |
| 3 | Parameter to sink method transact |
| 3 | Parameter to sink method update |
| 3 | Parameter to sink method write |
| 2 | Parameter type contains android.content.contentresolver |
| 2 | Parameter type contains android.content.context |
| 2 | Parameter type contains android.content.intent |

| | |
|---|---|
| 2 | Parameter type contains android.database.cursor |
| 2 | Parameter type contains android.filterfw.core.filtercontext |
| 2 | Parameter type contains android.net.uri |
| 2 | Parameter type contains com.android.inputmethod.keyboard.key |
| 2 | Parameter type contains com.google.common.io |
| 2 | Parameter type contains event |
| 2 | Parameter type contains java.io. |
| 2 | Parameter type contains java.io.filedescriptor |
| 2 | Parameter type contains java.lang.string |
| 2 | Parameter type contains observer |
| 2 | Parameter type contains writer |
| 1 | Permission name is ACCESS COARSE LOCATION |
| 1 | Permission name is ACCESS FINE LOCATION |
| 1 | Permission name is ACCESS LOCATION EXTRA COMMANDS |
| 1 | Permission name is ACCESS NETWORK STATE |
| 1 | Permission name is ACCESS WIFI STATE |
| 1 | Permission name is ADD VOICEMAIL |
| 1 | Permission name is AUTHENTICATE ACCOUNTS |
| 1 | Permission name is BACKUP |
| 1 | Permission name is BLUETOOTH |
| 1 | Permission name is BLUETOOTH ADMIN |
| 1 | Permission name is BROADCAST STICKY |
| 1 | Permission name is CALL PHONE |
| 1 | Permission name is CALL PRIVILEGED |
| 1 | Permission name is CAMERA |
| 1 | Permission name is CHANGE CONFIGURATION |
| 1 | Permission name is CHANGE NETWORK STATE |
| 1 | Permission name is CHANGE WIFI STATE |
| 1 | Permission name is CLEAR APP USER DATA |
| 1 | Permission name is DEVICE POWER |
| 1 | Permission name is DISABLE KEYGUARD |
| 1 | Permission name is DUMP |
| 1 | Permission name is GET ACCOUNTS |
| 1 | Permission name is GET TASKS |
| 1 | Permission name is GLOBAL SEARCH |
| 1 | Permission name is INTERNET |
| 1 | Permission name is KILL BACKGROUND PROCESSES |
| 1 | Permission name is MANAGE ACCOUNTS |
| 1 | Permission name is MANAGE APP TOKENS |

| | |
|---|---|
| 1 | Permission name is MODIFY AUDIO SETTINGS |
| 1 | Permission name is MODIFY PHONE STATE |
| 1 | Permission name is MOUNT UNMOUNT FILESYSTEMS |
| 1 | Permission name is NFC |
| 1 | Permission name is READ CALENDAR |
| 1 | Permission name is READ CALL LOG |
| 1 | Permission name is READ CONTACTS |
| 1 | Permission name is READ EXTERNAL STORAGE |
| 1 | Permission name is READ HISTORY BOOKMARKS |
| 1 | Permission name is READ PHONE STATE |
| 1 | Permission name is READ SMS |
| 1 | Permission name is READ SOCIAL STREAM |
| 1 | Permission name is READ SYNC SETTINGS |
| 1 | Permission name is READ SYNC STATS |
| 1 | Permission name is READ USER DICTIONARY |
| 1 | Permission name is REBOOT |
| 1 | Permission name is RECEIVE BOOT COMPLETED |
| 1 | Permission name is RECEIVE SMS |
| 1 | Permission name is RECORD AUDIO |
| 1 | Permission name is RESTART PACKAGES |
| 1 | Permission name is SEND SMS |
| 1 | Permission name is SET DEBUG APP |
| 1 | Permission name is SET TIME ZONE |
| 1 | Permission name is SET WALLPAPER |
| 1 | Permission name is SET WALLPAPER COMPONENT |
| 1 | Permission name is STOP APP SWITCHES |
| 1 | Permission name is SYSTEM ALERT WINDOW |
| 1 | Permission name is UPDATE DEVICE STATS |
| 1 | Permission name is USE CREDENTIALS |
| 1 | Permission name is USE SIP |
| 1 | Permission name is VIBRATE |
| 1 | Permission name is WAKE LOCK |
| 1 | Permission name is WRITE CALENDAR |
| 1 | Permission name is WRITE CONTACTS |
| 1 | Permission name is WRITE EXTERNAL STORAGE |
| 1 | Permission name is WRITE HISTORY BOOKMARKS |
| 1 | Permission name is WRITE SETTINGS |
| 1 | Permission name is WRITE SMS |
| 1 | Permission name is WRITE SOCIAL STREAM |

| | |
|---|---|
| 1 | Permission name is WRITE SYNC SETTINGS |
| 1 | Permission name is WRITE USER DICTIONARY |
| 2 | Return type is android.database.Cursor |
| 2 | Return type is android.net.Uri |
| 2 | Return type is android.os.Parcelable |
| 2 | Return type is boolean |
| 2 | Return type is byte[] |
| 2 | Return type is com.android.internal.telephony.Connection |
| 2 | Return type is int |
| 2 | Return type is java.util.List |
| 2 | Return type is java.util.Map |
| 2 | Return type is void |
| 3 | Value from method get to sink method |
| 3 | Value from method parameter to native method |
| 3 | Value from source method create to return |
| 3 | Value from source method get to return |
| 3 | Value from source method is to return |
| 3 | Value from source method obtainMessage to return |
| 3 | Value from source method query to return |
| 3 | Value from source method writeToParcel to return |
| 3 | Method starting with insert invoked |