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, esses dados seguem um fluxo específico indo de *data sources* para *data sinks*. O vazamento de dados acontece quando dados sensíveis chegam sem autorização em *sinks*, para prevenir isso, técnicas estáticas e dinâmicas de *Flow Enforcement* garantem que esses dados não cheguem nessas *sinks*. 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 mais utilizados e decidir quais os mais apropriados para esse problema.

Abstract

A program computes in either sensitive and non-sensitive data and follows a specific flow, from data sources to data sinks. 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 impractical 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.

List of Figures

List of Tables

List of Acronyms

Contents

1

Introduction

Every program computes on either sensitive data or non-sensitive data. Sensitive is any data that can be used to identify the user or any private user information, such as photos, International Mobile Equipment Identity (IMEI) nd biometric data. Non-sensitive data is any dynamic information that do not identify the user, oftenly this kind of information if public or shared, such as application source code.

In an application, 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).

Dynamic Flow Enforcement are techniques that tracks and enforces information flow 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.

Both Static and Dynamic Flow Enforcement techniques require information of which methods is a source of sensitive data and which is a data sink. This is used to identify if a sink method is truly leaking sensitive data or not. 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).

Considering that issue, Rasthofer et al. 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 Trees and Naive Bayes with the SVM and choosed 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

Background

3

Machine Learning Models

3.1 Monolithic

3.1.1 Decision Tree

3.1.2 K-Nearest Neighbors

3.1.3 Multi Layer Perceptron

3.1.4 Naive Bayes

3.1.5 Random Forests

3.1.6 Support Vector Machine

3.2 Dynamic Classifier Selection

3.2.1 KNORAE

3.2.2 KNORAU

3.2.3 META-DES

3.2.4 OLA

3.2.5 Single Best

3.2.6 Static Selection

4

Architecture

The system architecture is shown in figure 4.1, and the first step to classify Andoir methods is to have a collection of Android APIs. The APIs used in the work have been selected from two different repositories: the default repository of images used by RASTHOFER; ARZT; BODDEN (2014) and other repository is commonly used by mobile developers that need methods only available in APIs that have been 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, which is done by extracting APIs from real devices.

If the extraction is done from an emulator, the API comes without the methods fullt implemented, these methods are called method stubs. As long as you try to execute any of these methods, 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 small starting list of Android methods that have been hand classified, called hand annotated. With the methods and APIs in hand, the following step is to apply the feature extractor to each API, concatenate all the results and remove all the repeated entries. 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 5.1.

After the dataset creation, 30 other datasets are randomly sampled from the original with the objective to train and evaluate the classification algorithms. Each of these datasets are subdivided into train dataset and test dataset, containing 80% of the original dataset to train the model and 20% for test the model effectiveness. 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.

Each of the 30 datasets consists in an evaluation step, and after one step, the model is overwritten and a fresh one is used, forgetting all the information previously learned. In the evaluation step, the model is evaluated using precision, recall, F1 score and accuracy, these metrics are saved and at the end of all 30 steps, the mean and standard deviation is calculated and reported in the results, Section 5.4.

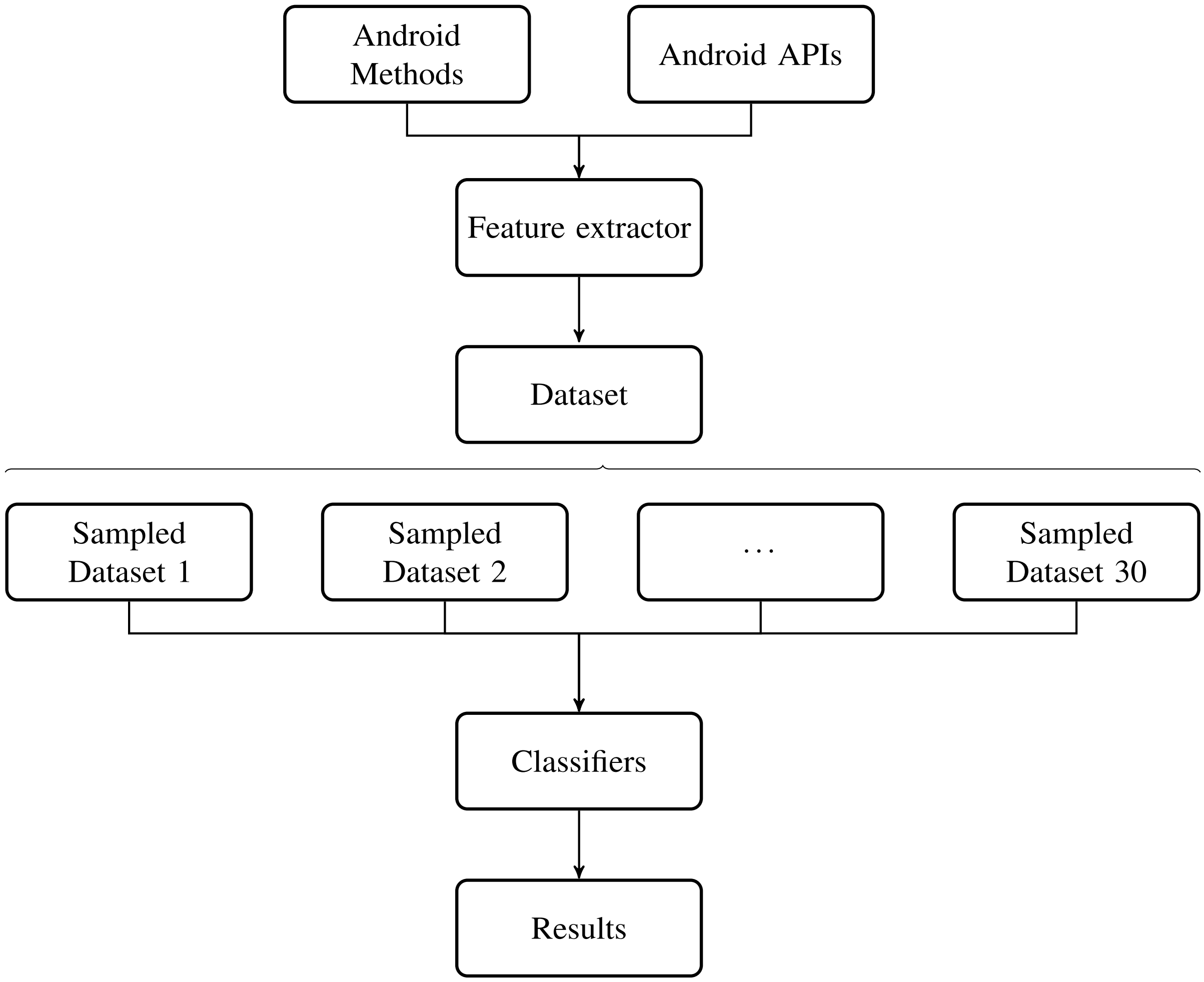

Figure 4.1: Overview of the proposed evaluation flow. First, the feature extractor uses Android Methods and Android APIs to create a dataset, which will be sampled into other 30 different datasets that will be used to evaluate the Classifiers.

5

Results

In this chapter, the results are exposed in the following way: Section 5.1 shows how the dataset has been created and the overall proportion of classes. Section 5.2 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 5.3 compares the KNORAE, KNORAU, META-DES, OLA, Single Best and Static Selection. After the comparison between Monolithic and MCS, the overall comparison is discussed in Section 5.4.

Using the dataset described in Section 5.1 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 5.2 and 5.3 to ease future replications of this work.

Each classifier is evaluated with 4 different metrics, precision (equation 5.1), recall (equation 5.2), F1 score (equation 5.3) and accuracy (equation 5.4). 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.

$$precision = \frac{TP}{TP+FP} \tag{5.1}$$

$$F1 = \frac{2 \times recall \times precision}{recall + precision} \tag{5.3}$$

$$recall = \frac{TP}{TP+FN} \tag{5.2}$$

$$accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{5.4}$$

5.1 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 5.1: The final dataset proportion without duplicated entries.

With that said, Table 5.1 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 5.1. 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.

```
False   False   False   False   False   True    False   True    False   False   False   True    False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   True    False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   True    False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   True    False   False   False   False   False   False   False   False   False
False   True    False   False   False   False   False   False   False   False   True    False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   True    False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   True    False   False   False   False   False   False
False   False   False   sink    '<com.android.internal.telephony.gsm.GsmSMSDispatcher: void dispatch
(android.content.Intentjava.lang.String)>'

False   False   False   False   False   True    False   True    False   False   False   True    False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   True    False   False   False   False   False   False   False   False   False
False   True    False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   True    False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   True    False   False   False   False   False   False   False   False   False
False   True    False   False   False   False   False   False   False   False   True    False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   False   False   False   False   False   False   False
False   False   True    False   False   False   False   False   False   False   False   False   False   False
False   False   False   False   False   False   False   True    False   False   False   False   False   False
False   False   False   sink    '<com.android.internal.telephony.gsm.GsmSMSDispatcher: void dispatch
(android.content.Intentjava.lang.String)>'
```

Figure 5.1: 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.

5.2 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 α 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 5.2, 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			
Linear SVM	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 5.2: Parameters for Monolithic Classifiers

Table 5.3 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.

5.3 Multiple Classifier System

For the Multiple Classifier System we used KNORAE, KNORAU, 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 algorithms we use the same base classifier configuration. For Perceptron we used no regularization,

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)
Linear 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 5.3: Result for Monolithic Classifiers, Mean (Standard Deviation), the best classifier for each metric is highlighted in bold. Using the

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 KNORAE and KNORAU 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 5.4.

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

Table 5.4: Parameters for Multiple Classifier System

In Table 5.5 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 KNORAE. Considering the mean and standard deviation, the results for KNORAE, KNORAU, 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 KNORAE, KNORAU and META-DES in Precision, Recall and F1. Comparing the Accuracy there is a smaller gap.

Model	Precision	Recall	F1 Score	Accuracy
KNORAE	0.8602 (0.0517)	0.8395 (0.0656)	0.8469 (0.0455)	**0.8580 (0.0306)**
KNORAU	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 5.5: 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

Model	Precision	Recall	F1 Score	Accuracy
KNORAE	0.8607 (0.0556)	**0.8368 (0.0678)**	**0.8452 (0.0471)**	**0.8570 (0.0273)**
KNORAU	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 5.6: 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

When using Perceptron as main classifier, Table 5.6, we can observe that KNORAE 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 KNORAE, KNORAU, META-DES, Static Selection and OLA are in the same interval for all the metrics. The difference between

KNORAE and META-DES is smaller in Precision, F1 and Accuracy but has a smaller Recall comparing to OLA.

The results for best classifier using Decision Tree has close results to the best classifier using Perceptron. Comparing KNORAE 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.

5.4 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 Android API methods categorization. Table 5.7 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 5.7: The best of Monolithic compared to the best of MCS, Mean (Standard Deviation). The best classifier for each metric is highlighted in bold.

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 5.8: C 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.

Looking deep to the results for each class, we can observe in Table 5.8 that the MLP has a better score in the majority of classes and metrics and METADES using Decision Tree as base classifier has better Precision when classifying the Source methods. If we look at the standard deviation, we can conclude that the classifiers are also in the same interval for all the metrics, this is another evidence that a simpler algorithm can address this problem.

6

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 METADES with Decision Tree and Linear SVM. These classification algorithms are at the same baseline value, the precision difference between MLP and METADES using Decision Tree is 0.011 and recall of just 0.0089, indicating that a simpler algorithm can solve the problem.

References

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

ANDROID Hidden APIs. Accessed: 01-11-2018.

AOSP Java Code Style for Contributors. Accessed: 12-10-2018.

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

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

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

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

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

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...** [S.l.: s.n.], 2014.

SASAKI, Y. et al. The truth of the F-measure. **Teach Tutor mater**, [S.l.], 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...** [S.l.: s.n.], 2017. p.485–490.

SUSI Android APIs. Accessed: 01-11-2018.

VALLÉE-RAI, R. et al. Optimizing Java bytecode using the Soot framework: is it feasible? In: INTERNATIONAL CONFERENCE ON COMPILER CONSTRUCTION. **Anais...** [S.l.: s.n.], 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**, [S.l.], v.1, n.1, p.56–71, 2010.

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