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

By

WALBER DE MACEDO RODRIGUES

B.Sc. Dissertation Proposal

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

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 described in section 3.1.1, K-Nearest Neighbors in section 3.1.2, Multi Layer Perceptron in section 3.1.3, Naive Bayes section 3.1.4 and Support Vector Machine in section 3.1.6. For Multiple Classifier System (MCS), the section 4.3 will compare the KNORAE 3.2.1, KNORAU 3.2.2, META-DES 3.2.3, OLA 3.2.4, Single Best 3.2.5 and Static Selection 3.2.6. After the comparison between Monolithic and MCS, the overall comparison is going to be 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 train and test, containing 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 is calculated and displayed in the result tables. For each classification method, both Monolithic and MCS, will have its setup described in section 4.2 and 4.3 to ease future replications of this work.

Each classifier will be evaluated with 4 different metrics, precision equation 4.1, recall equation 4.2 , F1 score equation 4.3 and accuracy equation 4.4. 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 true 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, a much 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, containing information about the method name, parameters, return type, method and classes modifiers, class modifiers, if exists data flow in the method return or parameters and the required permissions.

Lexical features follows the idea that the Android APIs adopt 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, already 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 semantic, syntactic and 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).

With that said, the table 4.1 shows the proportion of each class for the dataset, first we evaluated 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 all the APIs found, 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 if 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 neither nor. 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.

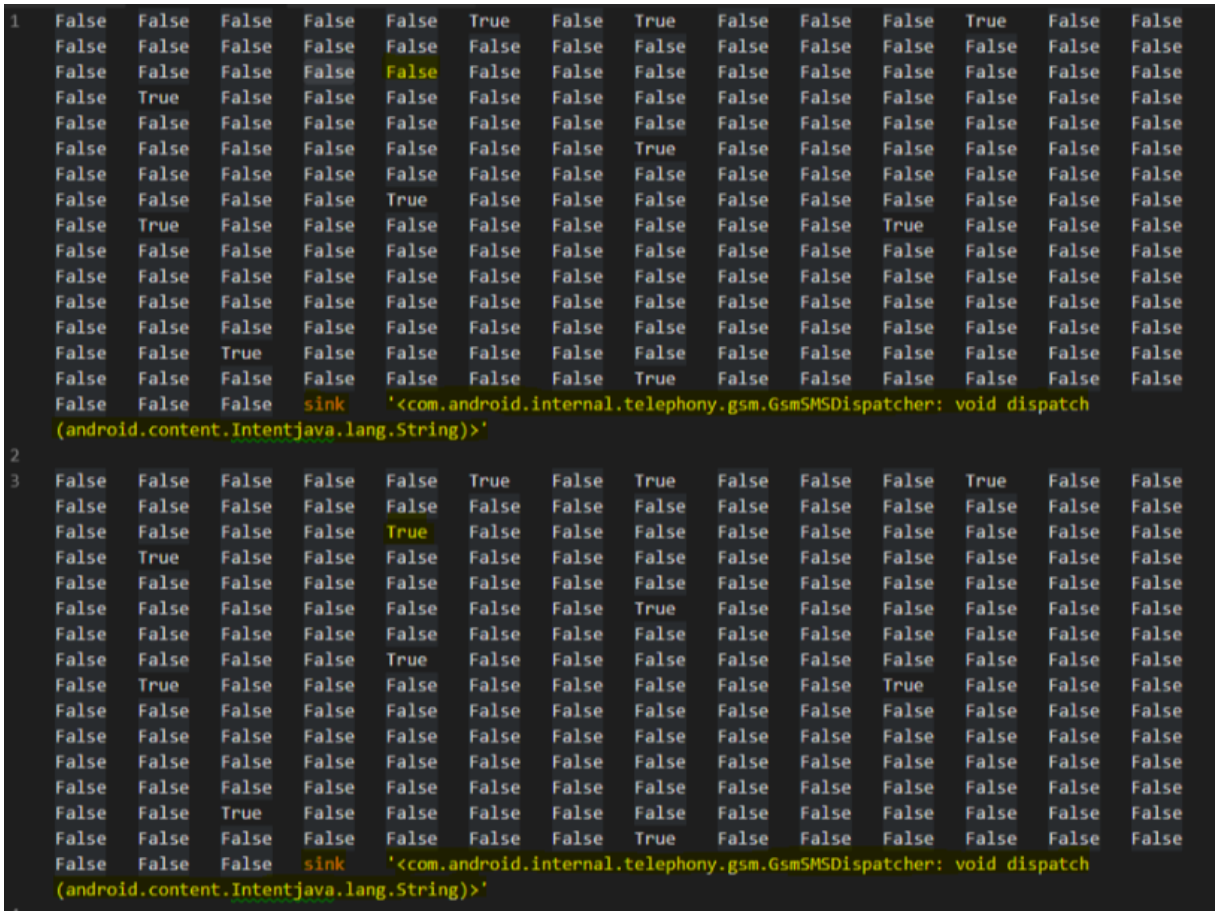

Figure 4.1: Methods through different API Levels. We can observe in yellow the feature "Method modifier is PUBLIC" and the full method name. Even for different features values, marked in orange, the method still belongs to the sink class.

API	Source	Sink	Neithernor	Total
API Level 17	316(59.0654)	131(24.4860)	88(16.4486)	535(100)
Dropped Names	322(59.3002)	134(24.6777)	87(16.0221)	543(100)
Dropped Entries	375(55.9701)	176(26.2687)	119(17.7612)	670(100)

Table 4.1: Overview of the generated dataset, Support(Percentage). First, the proportion for API Level 17, second, the dataset without duplicated method names (Dropped Names) and finally, the final dataset without duplicated entries (Dropped Entries).

4.2 Monolithic

For the Monolithic classifiers, we have the Decision Tree, discussed in section 3.1.1, Multinomial and Bernoulli Naive Bayes 3.1.4, K-Nearest Neighbor (KNN) discussed in section 3.1.2, Support Vector Machine (SVM) 3.1.6 and Random Forest 3.1.5.

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. For Perceptron we used no regularization, no early stopping, max of 1000 iterations and tolerance of 10^{-1}.

Model	Precision	Recall	F1 Score	Accuracy
Decision Tree	0.8474(0.0505)	0.8388(0.0549)	0.8410(0.0357)	0.8497(0.0217)
Multinomial NB	0.8297(0.0595)	0.8211(0.0552)	0.8231(0.0416)	0.8387(0.0282)
Bernoulli NB	0.8307(0.0609)	0.8180(0.0561)	0.8220(0.0424)	0.8378(0.0271)
KNN	0.8635(0.0547)	0.7991(0.0657)	0.8217(0.0484)	0.8432(0.0307)
Linear SVM	0.8920(0.0559)	0.8718(0.0539)	0.8796(0.0436)	0.8859(0.0284)
MLP	0.8936(0.0547)	**0.8759(0.0541)**	**0.8830(0.0437)**	**0.8908(0.0302)**
Random Forest	**0.8953(0.0568)**	0.8522(0.0605)	0.8684(0.0438)	0.8791(0.0309)

Table 4.2: Result for Monolithic Classifiers, Mean(Standard Deviation), the best classifier for each metric is highlighted in bold.

We can observe that the

4.3 Multiple Classifier System

For the Multiple Classifier System we used KNORAE, discussed in section 3.2.1, KNORAU in section 3.2.2, META-DES section 3.2.3, OLA in section 3.2.4, Single Best discussed in section 3.2.5 and Static Selection in 3.2.6. 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, 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 parameters were KNN to estimate the classifier competence using 7 neighbors, with no dynamic pruning and no indecision region, this set of parameters is used in KNORAE and KNORAU. 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.

In table 4.3 is presented the MCS algorithms using Decision Tree classifier. We can observe a better mean result using META-DES in all metrics. Considering the mean and standard

deviation, the results for KNORAE, OLA, Static Selection and META-DES are very close to the same interval.

When using Perceptron as main classifier, table 4.3, we can observe that KNORAE has the best mean results in every metric. But also, when considering the mean and standard deviation, the results are in the same range comparing KNORAE, KNORAU, OLA and Static Selection.Comparing both Decision Tree and Perceptron results in MCS, the results are very close, having 0.8665(0.0316) as F1 Score for the best Perceptron MCS (KNORAE) and 0.8741(0.0284) for the best Decision Tree MCS (META-DES), a difference of just 0.0086.

Model	Precision	Recall	F1 Score	Accuracy
KNORAE	0.8726(0.0539)	0.8461(0.0609)	0.8659(0.0292)	0.8560(0.0412)
KNORAU	0.8504(0.0635)	0.8162(0.0766)	0.8382(0.0322)	0.8284(0.0520)
META-DES	**0.8826(0.0549)**	**0.8577(0.0608)**	**0.8741(0.0284)**	**0.8672(0.0437)**
OLA	0.8448(0.0531)	0.8254(0.0596)	0.8428(0.0244)	0.8320(0.0409)
Single Best	0.7733(0.0931)	0.7413(0.1029)	0.7662(0.0511)	0.7500(0.0764)
Static Selection	0.8466(0.0661)	0.8095(0.0680)	0.8337(0.0352)	0.8232(0.0512)

Table 4.3: 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	**0.8759(0.0605)**	**0.8450(0.0596)**	**0.8665(0.0316)**	**0.8571(0.0471)**
KNORAU	0.8631(0.0589)	0.8287(0.0674)	0.8516(0.0295)	0.8415(0.0489)
OLA	0.8431(0.0621)	0.8234(0.0648)	0.8404(0.0361)	0.8306(0.0496)
Single Best	0.7999(0.0781)	0.7687(0.0910)	0.7912(0.0369)	0.7770(0.0520)
Static Selection	0.8667(0.0628)	0.8282(0.0660)	0.8537(0.0327)	0.8426(0.0490)

Table 4.4: 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

4.4 Final Considerations

5

Conclusion

References

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

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.

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.