4-2021

Machine learning based approaches towards robust Android malware detection

Jiayun XU
Singapore Management University

Machine Learning Based Approaches Towards Robust Android Malware Detection

Xu Jiayun

Singapore Management University

2021

Machine Learning Based Approaches Towards Robust Android Malware Detection

Xu Jiayun

Submitted to School of Computing and Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

Robert DENG Huijie (Supervisor / Chair)
Professor of Information Systems
Singapore Management University

Yingjiu LI (Co-Supervisor)
Ripple Professor
Department of Computer and Information Science
University of Oregon

Xuhua DING
Associate Professor of Information Systems
Singapore Management University

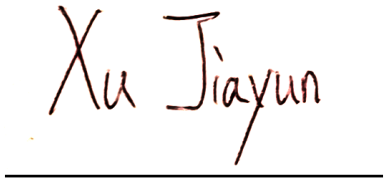
Debin GAO
Associate Professor of Information Systems
Singapore Management University

Jianying ZHOU (External Reviewer)
Professor of Information Systems Technology and Design
Singapore University of Technology and Design

Singapore Management University
2021

I hereby declare that this dissertation is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in this dissertation.

This PhD dissertation has also not been submitted for any degree in any university previously.



Xu Jiayun

05 February 2021

Machine Learning Based Approaches Towards Robust Android Malware Detection

Xu Jiayun

Abstract

The Android platform is becoming increasingly popular and numerous applications (apps) have been developed by organizations to meet the ever increasing market demand over years. Naturally, security and privacy concerns on Android apps have grabbed considerable attention from both academic and industrial communities. Many approaches have been proposed to detect Android malware in different ways so far, and most of them produce satisfactory performance under the given Android environment settings and labelled samples. However, existing approaches suffer the following robustness problems:

- In many Android malware detection approaches, specific API calls are used to build the feature sets, and their feature sets are fixed once the model has been trained. However, such feature sets lack of robustness against the change of available APIs. Since there are always new APIs released with old ones deprecated [1] during the evolvement of Android specifications. If developers switch from old APIs to new ones in app development, older Android malware detection models which are trained before the release of new APIs may not be effective then, because these new APIs are not included in the previously fixed feature sets.

- Besides, existing approaches are also lack of robustness towards the label noises. Recent research discovered that sample labels provided by malware detection websites may not be always reliable [43], and we also figure out that 10% of sample labels provided by VirusTotal change during a period of 2 years in our experiments. This indicated label noises cannot

be ignored in the training of Android malware detection models, while existing approaches which directly use the provided labels will suffer from the label noise problem.

- Furthermore, even if the sample labels are correct, there may still exist inconsistencies between the sample labels and the generated feature vectors in dynamic-based Android malware detection approaches. Since no triggering modules can perfectly trigger all potential malicious behaviors, and anti-analysis techniques are common in the apps. In this case, the triggered behavior traces collected from samples labelled as "malware" may not contain "malicious" behaviors, thus feature vectors built from such traces may become noises in the model training.

Towards the above problems, three different works are presented in this dissertation to provide robustness to Android malware detection in different ways:

The first work in this dissertation proposes a slow-aging Android malware detection solution named SDAC. Towards solving the model aging problem, SDAC evolves its feature set effectively by evaluating new APIs' contributions to malware detection using existing APIs' contributions. In detail, SDAC evaluates the contributions of APIs using their contexts in the API call sequences. These sequences are extracted from Android apps demonstrating how the APIs are used in real world cases. Based on these sequences, an embedding algorithm named API2Vec is deployed to map APIs into a vector space in which the differences among API vectors are regarded as the semantic distances. Then SDAC clusters all these APIs based on the semantic distances among them to create a feature set in the training phase, and extends the feature set to include all new APIs in the detecting phase. By the feature extension, SDAC can adapt to the changes in Android specifications and thus produces a robust approach against changes in Android OS specifications.

The second work in this dissertation is named Differential Training, which is a general framework designed to reduce the noise level of training data for any machine learning-based Android malware detection approach. We discover that labels of samples provided by Anti-Virus organizations change over time. The changes imply certain labels are erroneous, and thus distort the performance when such labels are used in training Android malware detection models. Differential Training, which functions as a general framework, can detect label noises with different Android malware detection approaches. For the input sample apps, Differential Training firstly generates the noise detection feature vectors from all the intermediate states of two identical deep learning classification models. Then it applies outlier detection algorithms on these noise detection feature vectors, and the outliers detected are regarded as coming from noises. With the label noises being detected and reduced, Differential Training can thus help improve the detection accuracy of Android malware detection approaches.

The third work in the dissertation is a noise-tolerant dynamic-based Android malware detection approach named Dynamic Attention. In dynamic-based Android malware detection approaches, the triggered behavior traces collected from samples with "malware" labels may not contain "malicious" behaviors due to the imperfect trigger procedure or anti-analysis methods, so they are in fact mislabelled when used in training Android malware detection models. Dynamic Attention is thus designed to solve this mislabelling problem: it identifies the label noises based on the variances of the attention weights associated within the behavior traces derived from malicious apps, and assigns correctly-labelled behavior traces with high weights and wrongly-labelled ones with low weights during the model training. By doing so, Dynamic Attention makes the classification model learn less from wrongly-labelled feature vectors and gains resistances against the noises. This approach also enjoys high practicality, since it relies on neither domain knowledge nor manual inspection in the model

training.

This dissertation contributes to the robustness of Android malware detection approaches in various ways. In particular, SDAC is robust towards changes in Android specifications, Differential Training provides robustness against label noises for Android malware detection in static analysis, and Dynamic Attention achieves the same goal for Android malware detection in dynamic analysis.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to firstly thank my supervisors Prof. Robert Deng and Prof. Yingjiu Li, for their support during my PhD study. Since the start of the programme, Prof. Deng and Prof. Li gave me valuable advice and guidance which helped me conquer the difficulties in both my research and my life. I offer my sincere appreciation and gratitude for their constant encouragement, extraordinary patience, advice of great value, edits on my writing, and funding throughout all the stages over all these years. To my other dissertation committee members: Prof. Xuhua Ding, Prof. Debin Gao, and Prof. Jianying Zhou, I am also very grateful for their valuable help in validating my research, thanks a lot for your support.

Then I would like to acknowledge my classmates in the SIS research lab: Daoyuan Wu, Ke Xu, and Ximing Liu, for their assistance on my research. They graciously make invaluable assistance, considerable comments, and suggestive discussion to the outline of my research and this final dissertation.

Finally, I would like to express my gratitude to my parents for their understanding and love over these years. Thank you for doing everything possible to put me on the path of my abroad studying and consistently encouraging me to be brave and optimistic towards the challenges. Thank you.

Chapter 1

Introduction

Android malware detection has long been an interest of researches for both academic and industrial communities and many approaches are proposed to reduce the threat caused by malware. Existing approaches usually assume that no changes on the sample labels or Android OS specifications will take place during their life periods. However, in real world cases, these changes can not be ignored: For example, changes on Android OS specifications will lead to model aging due to the introducing of new APIs, while the labels of samples may also change after the detection models are trained. These unconsidered changes deprave the performance of detection models.

1.1 SDAC: A Slow-Aging Solution for Android Malware Detection Using Semantic Distance Based API Clustering

The first work in this dissertation introduces a novel slow-aging solution named SDAC, which is proposed to address the model aging problem in Android malware detection. The model aging is due to the lack of adapting changes in Android specifications. Different from periodic retraining of detection

models in existing solutions, SDAC evolves effectively by comparing new APIs' contributions to malicious behaviors with those of existing APIs.

In SDAC, the contributions of APIs are evaluated by their contexts in the API call sequences extracted from Android apps. A neural network is applied to the sequences to assign APIs to vectors, among which the differences of API vectors are regarded as the semantic distances among the APIs. SDAC then clusters all APIs based on their semantic distances to create a feature set in the training phase, and extends the feature set to include all new APIs in the detecting phase. Without being trained by any labelled new apps, SDAC can adapt to the changes in Android specifications by simply identifying new APIs appearing in the detection phase. In extensive experiments with datasets dated from 2011 to 2016, SDAC achieves significantly higher accuracy and lower aging speed compared with MaMaDroid, a state-of-the-art Android malware detection solution which maintains resilience to API changes.

1.2 Differential Training: A Generic Framework to Reduce Label Noises for Android Malware Detection

The second work in this dissertation aims at solving the label noise problem in machine-learning based Android malware detection approaches. That is, the training data may contain wrong labels and it is challenging to make the training data noise-free at a large scale. To address this problem, we propose a generic framework named Differential Training, to reduce the noise level of training data for any machine learning-based Android malware detection.

Our framework makes use of all intermediate states of two identical deep learning classification models during their training with a given noisy training dataset and generates a noise-detection feature vector for each input sample.

The framework then applies a set of outlier detection algorithms on all noise-detection feature vectors to reduce the noise level of the given training data before feeding it to any machine learning-based Android malware detection approach. In our experiments with three different Android malware detection approaches, our framework can detect significant portions of wrong labels in different training datasets at different noise ratios, and improve the performance of Android malware detection approaches.

1.3 Dynamic Attention: A Noise-Tolerant Dynamic Analysis Approach to Android Malware Detection based on Attention Variances

Dynamic Attention, as the third work in the dissertation, solves a common problem in dynamic-based Android malware detection solutions. In dynamic-based Android malware detection approaches, behavior traces triggered from malicious Android apps often are assigned with the same label "malicious" as these apps. However, such labels can be erroneous, because (1) the malicious behaviors of the malicious apps may not be triggered due to imperfect trigger procedures or anti-analysis methods deployed in malware, and (2) it is impossible even for domain experts to manually verify the "malicious" labels since the triggered behavior traces are usually lengthy and the techniques for composing malware are highly complicated and constantly evolving.

Dynamic Attention is thus proposed to address this problem. In detail, Dynamic Attention identifies wrong "malicious" labels by examining the variances of the attention weights associated with the behavior traces that are derived from malicious apps. After identifying the behavior traces with wrong "malicious" labels, Dynamic Attention assigns correctly-labelled behavior traces with high weights and wrongly-labelled ones with low weights during the

model training. Consequently, the classification model trained is significantly more tolerant to noisy "malicious" labels than the vanilla classification model that is trained without weight assigning.

1.4 Contributions and Organization

To summarize, the following contributions have been made in this dissertation:

- We designed a novel slowing aging solution named SDAC for Android malware detection. SDAC performs slow-aging by mapping new APIs' contribution to malicious behaviors to evaluated old ones, thus allows previously-trained models to effectively evaluate those new APIs brought by Android specification changes. The best versions of SDAC achieve both high accuracy with average F-score 97.49%, and slow aging speed with average F-score decline 0.11% per year over five years in our experiments. The other versions have lower requirements on computing resources, but still perform better than the state of the art.

- We develop a new generic framework, Differential Training, to reduce label noises for large-scale Android malware detection. Differential Training employs a novel approach to detecting noisy labels in multiple iterations according to the intermediate states of two deep learning classification models of identical architecture, one of which is trained on the whole training set of apps, and the other is trained on a randomly down-sampled set of apps. A new heuristic is proposed to distinguish between wrongly-labeled apps and correctly-labeled apps based on an outlier detection on their loss values, which are taken from the intermediate states of the two classification models.

- We develop Dynamic Attention, a noise-tolerant dynamic analysis approach to Android malware detection with the help of attention

mechanism. Dynamic Attention performs noise-tolerant detection by making the model learn less from potentially noisy samples during the model training. In the sample weighting, Dynamic detection firstly calculate the self-attention weight variance for the values in the feature vector for each app in the training set, then it assigns high (low, respectively) weights to the feature vectors that are extracted from malicious apps if their attention variances are high (low, respectively). In this case, the model will learn less from the noisily-labelled dynamic behavior traces and gain resistances against the noises. To the best of our knowledge, Dynamic Attention is the first noise-tolerant dynamic analysis approach to Android malware detection.

The remainder of this dissertation is organized as follows: Chapter 2 is a literature review which examines closely related research on Android malware detection and label noises detection. Chapter 3 proposes the slow-aging Android malware detection model "SDAC". Chapter 4 describes Differential Training, the general framework which can work with various kinds of Android malware detection approaches to reduce noises. And Chapter 5 introduces Dynamic Attention, the noise-tolerant dynamic analysis approach on Android malware detection which makes use of attention mechanism. Finally, Chapter 6 summarizes the contributions of this dissertation.

Chapter 2

Literature Review

In this chapter, we firstly demonstrate the related works about the evolvement of Android framework, including the investigation of Android framework evolvement and how it will affect the applications and developers in practice. Note that most of them focused on how to improve the application usability rather than Android malware detection among these works, while no rigorous study has been conducted on the topic of Android malware detection. Then we demonstrate existing research on the label noise problem in Machine learning, showing a brief view on the mechanisms proposed to alleviate the negative effects caused by wrongly-labelled samples in the model training. After that, a survey on the current Android malware detection approaches is presented, in which these approaches are categorized by different standards such as analysis type, signature-based or learning-based, and features of different kinds. From the brief survey, it is shown that existing approaches are lack of both the robustness towards Android specification evolvement and the robustness towards the label noises, which are the goals in the dissertation.

Android Framework Evolvement. Android apps rely on Android APIs to perform their functions, and many APIs are added or deprecated in Android specifications over time. The impact of API evolution to the usability of apps has been studied recently. For example, McDonnell, Ray, and Kim investigated how

Android app developers follow and adopt Android API changes over time [54]. Linares-Vásquez et al. studied the relationship between API changes and fault proneness, and evaluated its threat to the success of Android apps [49]. Brito et al. studied the adoption of API deprecation messages and its impact on software evolution [23]. Recently, Wu et al. focused on inconsistency between the versions of declared Android API frameworks and the actual ones used in Android apps [85]. While most of the previous research in this area focused on the usability of apps, no rigorous study has been conducted on the impact of Android framework evolvement on malware detection.

The Label Noise Problem in Machine Learning.

The label noise problem has been recently addressed in the machine learning literature, where the focus is on how to train classification models that are tolerant to label noises. Various approaches have been developed to alleviate the negative effects of wrongly-labeled samples in model training to improve the quality of the finally-trained models.

One approach adjusts the loss calculation in the process of model training according to label noise estimation [60, 98]. Another approach relies on the models of special structure that can reduce the impact caused by label noise in model training [74, 63]. And other approaches aim at training noise-tolerant models for various purposes [37, 56, 82]. All of these approaches perform noise detection according to the final states of input samples in either the training phase or the testing phase.

According to Schein, et al., the intermediate states of an input sample are useful in measuring the uncertainty between the predicted label and the actual label of the sample in the process of model training [70]. Inspired by this, Chang et al. accelerated the process of model training [25]. However, the intermediate states of input samples during model training have not been utilized to process noisy samples except in Co-Teaching [39], where the individual loss value of each input sample in each mini-batch is examined during model training.

Android Malware Detection. Android malware detection can be categorized into static analysis and dynamic analysis (e.g., [91, 33, 26, 46, 71, 14]). Static analysis detects Android malware according to the information extracted from app APK files. It can be further categorized into signature-based solutions (e.g., [28, 34, 36]) and learning-based solutions. We briefly summarize some learning based solutions that are more closely related to SDAC than other solutions.

A wide variety of features have been examined in developing learning based solutions. For example, Arp et al. devised Drebin to extract eight classes of features (e.g., network addresses, component names, permissions, and API calls) from manifest files and disassembled codes. Avdiienko et al. examined the difference in sensitive data flows between malware and benignware [21]. Yang et al. designed DroidMiner to extract malicious behavior patterns from APIs and framework resources [92]. In another work, Ke, Li, and Deng devised ICCDetector to extract inter-component communication features from app components [89].

In addition, DroidAPIMiner proposed by Aafer, Du, and Yin extracts a set of API-level features including critical API call frequencies, framework classes, and API parameters [13]. DroidSIFT proposed by Zhang et al. extracts weighted contextual API dependency graphs from apps based on sensitive APIs [97]. MAST proposed by Chakradeo et al. examined strong relationships between declared indicators of application functionality (e.g., permissions, intent filters, and the presence of native code) [24]. Many other types of features are also used in learning-based solutions (e.g., [40, 96, 41, 93, 50, 27]).

A common feature of the feature sets in most learning-based solutions is that they are "static", not keeping up with the evolvement of Android frameworks. Consequently, the accuracy of such solutions may decline significantly over time (i.e., model aging), which has been observed in both industry (e.g., [80]) and academia (e.g., [99, 69]) recently.

Chapter 3

SDAC: A Slow-Aging Solution for Android Malware Detection Using Semantic Distance Based API Clustering

3.1 Introduction

Most Android malware detection models age quickly. According to Zhu and Dumitras [99], an Android malware detection model generated in 2012 failed to detect any malware in the Gappusin family while a 2014 model could detect most of them. In another research of Wang from Baidu [80], the recall rate of an Android malware detection model developed at Baidu decreased by 7.6% in the first six months. Recent research has identified a main reason of model aging to be API changes over time in Android specifications [99, 69]. Apparently, malware samples making use of newly added APIs in performing malicious behaviors may evade from the detection of aged models.

The common solution to address the aging of Android malware detection models is either to update signature databases for signature-based malware

detection, or to renew malware detection models using new Android apps with true labels (i.e., malware and benignware) for learning-based models. However, this process is usually time-consuming and costly, which may involve many domain experts' efforts on the sample labelling and data sharing across multiple organizations. Furthermore, the true labels of newly collected apps may not be connivently or promptly available and even be mistaken in real life.

For instance, we downloaded the reports for a set of 42808 apps from VirusTotal[1] in July 2017 and July 2018 respectively. In these apps, about 11% (4717/42808) of them which were labelled as "benign" by all the antivirus engines in July 2017 turned out to be labelled as "dangerous" by at least one antivirus engine in July 2018, indicating that the labels may be erroneous for a long time. It is thus imperative to develop a slow-aging solution that remains accurate in malware detection for longer time and can be renewed without relying on the true labels of new apps.

While Android API changes have been identified as a major problem leading to model aging in Android malware detection [99, 69], the adaptation to API changes has not been rigorously addressed in the design of slow-aging solutions. A recent solution named PikaDroid [15] addressed the aging problem by utilizing the contextual information of sensitive APIs in malware detection. However, it does not adapt to the changes of Android specifications since the feature set in PikaDroid remains unchanged in its design.

Another approach, MaMaDroid [53] proposed a detection method which is resilient to the changes in Android specifications. In particular, MaMaDroid first abstracts application programming interfaces (APIs) to their corresponding packages (or package families) in the API execution paths derived from each Android app. It then summarizes all abstracted paths to a Markov model, and converts the Markov model to a feature vector for each app in model training and testing.

[1] VirusTotal is a website which aggregates multiple antivirus scan engines.

By abstracting APIs to packages, MaMaDroid is resilient to the adding of new APIs to *existing packages* and performs significantly better than other solutions such as DroidAPIMiner for Android malware detection. However, MaMaDroid does not address the contribution of any *new packages* to malware detection since the transitions caused by any new packages in Markov models convert to no features in MaMaDroid.

On average, about 340 new APIs in 4 new packages were added to each API-level compared to its previous one according to the Android developer documentation [1]. These new packages and new APIs are important factors leading to model aging in Android malware detection. Without model updating, the performance of existing malware detection models, including MaMaDroid, may downgrade significantly over time as more and more new packages and APIs are added in Android specifications and used in Android app development.

In this chapter, we develop a learning-based and slow-aging solution for Android malware detection. Our slow-aging solution is named SDAC, which stands for semantic distance based API clustering. SDAC identifies an API's contribution to malicious behaviors based on the *APIś contexts*, which refers to the APIs within a fixed-size window from the API in the API call sequences performed by apps. In particular, given a training set of apps with true labels, SDAC extracts API call sequences from the apps. Based on the extracted API sequences, SDAC applies a two-layer neural network to embed APIs into *API vectors* and arrange these vectors into a vector space. In the vector space, the APIs sharing common *API contexts* are located close to each other. A *feature set* is formed according to API vector clusters, where each feature is defined as the set of all APIs whose corresponding *API vectors* are in a same cluster.

For each app in the training set, a binary feature vector is generated by a one-to-one mapping from the feature set. An element in an app's feature vector is assigned to zero if none of the APIs in the mapped feature is used by this app, and it is one otherwise. Any classification models can be built based on

the feature vectors that are derived from training apps and their corresponding labels.

To make SDAC slow-aging, it is important to identify the new APIs' contributions to malware detection without knowing the true labels of the new apps that use these APIs. The key technique of performing such identification in SDAC is *feature extension*. In this step, each new API that appears in the testing set of apps is added to the closest feature in the feature set according to the distance measured in the API vector space. A new API's contribution to malware detection is modelled to be equivalent to the contribution of the other APIs in the same feature. A trained classification model does not need to be re-trained in the test phase since the contributions of all "old" APIs have already been evaluated in the training phase.

When performing malware detection over time on a series of testing sets in which apps are developed in successive time periods, SDAC can be executed in two major modes, SDAC-FEO and SDAC-FMU. SDAC-FEO is short for "SDAC-Feature Extension Only", in which feature extension is performed each time with a new testing set, while the trained classification model keeps unchanged all the time. In SDAC-FMU, which is short for "SDAC-Feature and Model Updating", both the classification model and the feature set are updated with new testing set. Note that although the classifiers are changed in some cases, both SDAC-FEO and SDAC-FMU do not need any labelled new samples in the successive time periods while can still keep a high accuracy over time, thus the whole solution SDAC is regarded as being slow-aging.

Both SDAC-FEO and SDAC-FMU require the current testing set to be wholly available in feature extension to collect the new APIs' contexts. To relax this constraint, we design SDAC-FEO-OL and SDAC-FMU-OL for online detection of individual apps without waiting for the whole testing set being available. In particular, they use existing classification models with no feature extension for online detection of individual apps, while after the whole testing

set is available, they resort to their off-line versions to update themselves.

We evaluated the performances of SDAC in different modes and versions using 70,142 Android app samples dated from 2011 to 2016. For simplicity, we refer to the scenario as our "default setting" in which the 2011 samples are used for both training and 5-fold cross validation, while the 2012-2016 samples are used for testing. The evaluation results in the default setting show that the F-score of SDAC-FEO declines by 4.81% per year on average from 2011 to 2016, while MaMaDroid declines by 7.67% per year. The average F-score of SDAC-FEO on these testing sets is 87.23%, which is higher than that of MaMaDroid in the same case(59.03%), by 28.2%. Compared to SDAC-FEO, SDAC-FMU further reduces the aging speed from 4.81% to 0.10% per year on average, and increases the average F-score from 87.23% to 97.09%. While the online versions are more efficient in classifying each app online, their accuracies are slightly lower, and their aging speeds are slightly higher than the corresponding non-online versions.

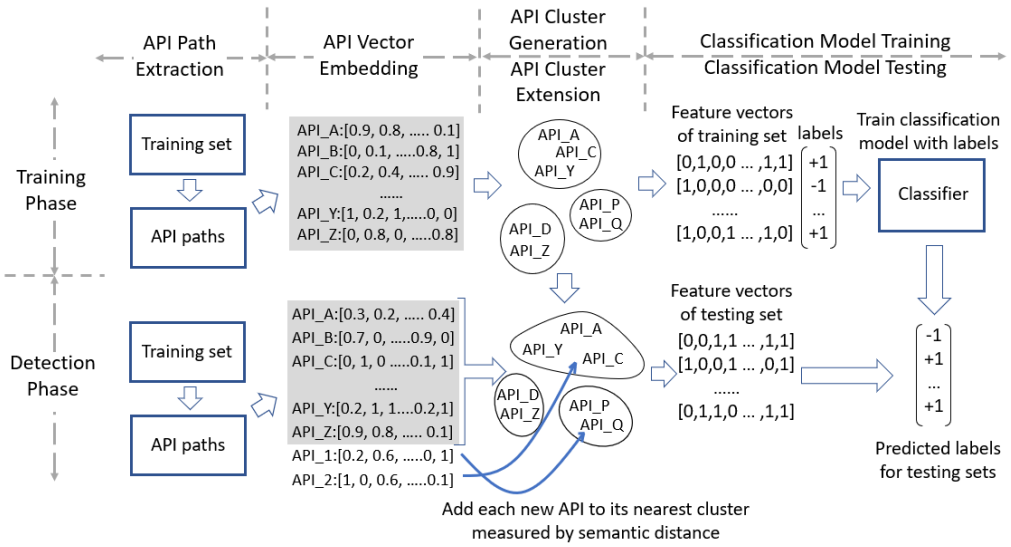


Figure 3.1: Structure of Basic SDAC with One Classification Model

3.2 Basic SDAC

An Android app can be considered as a set of operation paths, where each operation path is a sequence of operations that can be executed on Android platforms under certain conditions. In the design of SDAC, we focus on API call operations by which an Android app accesses Android system services and resources. An Android app may use any API that is provided in Android specifications at the time when it is developed. While Android specifications evolve over time from API-level 1 to API-level 27, a number of new APIs are added continually. Of course, an Android app cannot use any new APIs that do not exist at the time when it is developed.

Assumptions. SDAC is trained with a set of Android apps that are associated with true labels, including malware and benignware, where the apps in the training set are developed in a same time period. After training, SDAC is used to classify all apps in testing sets in other time periods. It is assumed that no true labels are available for the apps in testing sets when SDAC is in use.

A basic SDAC is first developed to classify apps in one testing set, which is developed in a time period after the training set. It is then extended in different modes and versions to classify apps in multiple testing sets, which are developed in successive later time periods.

Structure. The structure of basic SDAC is illustrated in Fig. 3.1. It consists of two phases, a *training phase* in which SDAC is trained with a training set, and a *detection phase* in which SDAC is used to detect malware in a testing set. In both phases, the basic SDAC proceeds through four steps, where the first two steps, including *API path extraction* and *API vector embedding*, are shared in both phases. The last two steps are *API cluster generation* and *classification model training* in the training phase, and *API cluster extension* and *classification model testing* in the detection phase.

3.2.1 API Path Extraction

The first step of SDAC is to extract a set of API call paths from each app it processes. A suitable static analysis tool such as FlowDroid [20] can be exploited to transform each app from bytecode to proper representation (e.g., Jimple code), and extract a directed call graph among its program methods. From a call graph, SDAC extracts a call graph snippet for each method, and excludes any call graph snippet that is a sub-graph of another call graph snippet.

The call graph snippet for a target method consists of all methods and all directed links between them in the call graph. In a call graph snippet, all the methods are located at most d links away from the target one in the call graph, where the links pointing to any methods that use no APIs are not counted in the snippet. SDAC then derives all API call paths from each call graph snippet.

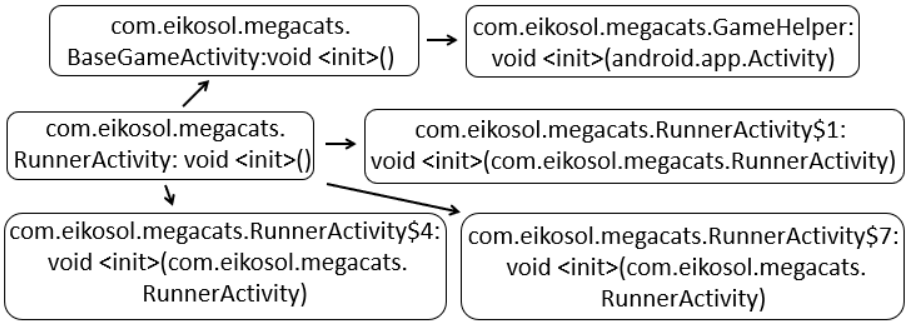


Figure 3.2: A Call Graph Snippet of "Mega Cats"

An Android game "Mega Cats" in Jimple code is used as an example to further clarify the API path extraction step. Fig. 3.2 shows a call graph snippet of Mega Cats with $d = 2$, where each block refers to a program class, and each directed link represents a call relationship between a caller method and a callee method.

Fig. 3.3 shows how SDAC builds an API call path from the call graph snippet. In particular, SDAC starts from the first line of Jimple code in method *com.eikosol.megacats.RunnerActivity:void<init>()* and records every invocation calls. If an invocation call points to another method,

SDAC jumps to the callee method and continues recording these invocation calls in that method until either it jumps again, or it reaches the last line of that method and then returns to the jumping point. In both cases, SDAC continues this recursive process until it ends up in method *com.eikosol.megacats.RunnerActivity:void<init>()*. The invocation calls that are recorded in this example are numbered in a sequential order from (1) to (12) in Fig. 3.3. SDAC generates an API call path from the recorded invocation call sequence by removing non-API calls.

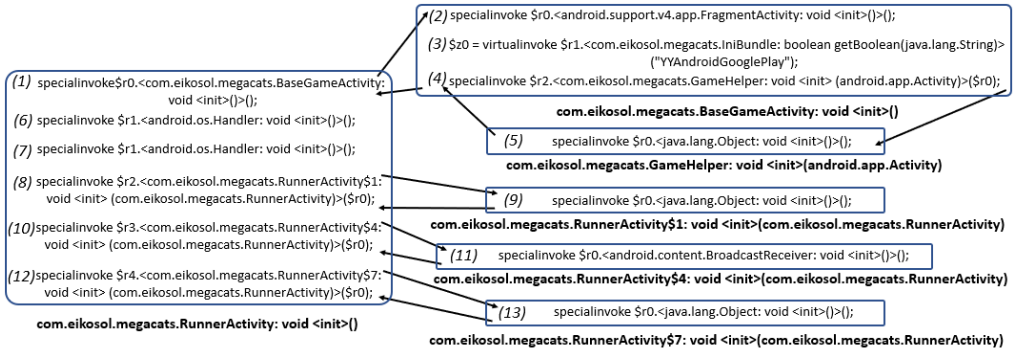


Figure 3.3: An Invoke Call Path of "Mega Cats"

If d is large enough, SDAC would output all possible API call paths for each app, which is an intractable number to program size [95]. To make SDAC practical, the parameter d is chosen to be relatively small, which is equal to the window size S as mentioned in the next subsection.

3.2.2 API Vector Embedding

The goal of this step is to embed each API into an n-dimensional real-valued vector in $[0, 1]^K$ according to the API's context in a set of apps. An *API's context* in a set of apps is defined to be the set of all its neighboring APIs within a fix-sized window containing the API in all API call paths that are extracted from the set of apps. Different APIs with similar contexts are mapped close to each other in the vector space.

The process of API vector embedding is illustrated in Fig. 3.4. Based on

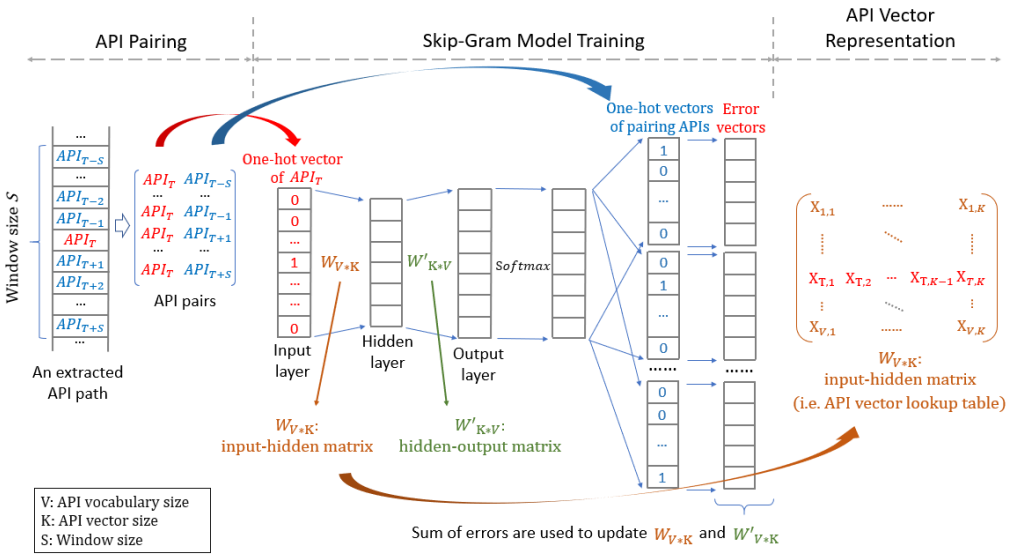


Figure 3.4: API Vector Embedding

the API call paths extracted in step "API path extraction", SDAC builds an *API vocabulary* which consists of all unique APIs in these paths. For each target API in the API vocabulary and for each extracted API path, SDAC derives all neighboring APIs which are at most S APIs away from the target API in the API path, where S is called window size. Then, SDAC pairs each target API with each of its neighboring APIs, and uses all such pairs to train a Skip-Gram model, which is a neural network with an input layer, a hidden layer, and an output layer [57]. In the model, a real-valued *input-hidden matrix* W_{V*K} is used to transform an input vector to a hidden input vector, and a real-valued *hidden-output matrix* W'_{K*V} is used to further transform a hidden input vector to an output vector, where V is the API vocabulary size, and K is the API vector size.

In the training process, a target API and all its paired APIs are regarded as the input and the target outputs of the Skip-Gram model, respectively. For a target API, the input of Skip-Gram model is a one-hot encoded vector of size V, where the index corresponding to the target API points to one and all other indexes point to zero. Similarly, a target output for a paired API is a one-hot encoded vector of size V where the index corresponding to the paired API points to one and all other indexes point to zero. For each target API, the Skip-Gram model

computes an output vector from the input vector by transforming it with the input-hidden matrix, the hidden-output matrix, and the softmax function (i.e., normalized exponential function), successively. Then, the Skip-Gram model updates the elements in these two matrices using backpropagation, which is a common optimization step for supervised learning of neural networks, so as to minimize the total error between the computed output and the target outputs.

After the Skip-Gram model is trained with all API pairs, SDAC outputs an API vector for each API. In Fig. 3.4, the API_T's vector is the T-th row in the input-hidden matrix, where T is the index of API_T in the API vocabulary. An API's vector embeds the API's context, which represents all its neighboring APIs that are encoded in the target outputs. If two different APIs are embedded to similar API vectors, thus they have similar contexts because the computed outputs are optimized in model training to approximate their target outputs.

3.2.3 API Cluster Generation and Extension

The third step of basic SDAC is API cluster generation in its training phase, and API cluster extension in its detection phase. The purpose of this step is to group the APIs in the API vocabulary into a number of clusters based on the *semantic distance* between APIs, where the semantic distance is defined as the Euclidean distance in the API vector space. The output of this step, which is a set of API clusters, will be used as the feature set for generating a feature vector for each app in the next step.

API Cluster Generation. In the training phase, SDAC applies the same-size k-means cluster algorithm from [35] to the API vectors that are calculated from the training set. The algorithm partitions API vectors into k clusters in which each API vector belongs to the cluster with nearest mean, where the difference in size among all clusters is at most one. Using the clusters, an *initial feature set* is defined by a one-to-one mapping from the clusters, where each feature

consists of all APIs whose API vectors appear in a same cluster. The APIs in a same feature share similar contexts (due to the closeness of their API vectors), and thus make similar contributions to malware detection in our model.

The same-size k-mean algorithm is chosen in SDAC because it can effectively avoid skew clustering results and thus maximize the differences between the feature vectors of benign apps and malicious ones. As described in [35], the same-size k-means cluster algorithm works as follows: Starting from k random initial means in the API vector space, the cluster algorithm generates initial k clusters from n API vectors by executing the following two steps recursively: (i) order API vectors by the distances between each one's nearest cluster and farthest cluster, and (ii) assign sorted API vectors to their nearest cluster until this cluster is full (number of vectors in this cluster equals to $\lfloor n/k \rfloor$), such full cluster will not be taken into consideration in forming the forthcoming clusters. After this initialization, the cluster algorithm runs through an iteration process to adjust the clustering results so as to reduce the variance of clusters until they converge to local optima.

API Cluster Extension. In the detection phase, SDAC outputs a set of API vectors derived from a testing set in the previous step. Now it extends the initial feature set to include all new APIs that appear in the API vocabulary of the testing set, but not in that of the training set. This step is also named *feature extension* because each cluster is regarded as one feature in the feature set.

In general, SDAC extends feature X to include a new API_Y if API_Y has the least average semantic distance from the APIs of feature X among all features, where semantic distance is measured by API vectors derived from **the testing set**. In a feature, if an API does not exist in any apps of the testing set, SDAC excludes it from the calculation of average semantic distance unless no API in the feature exists in the testing set, in which case corresponding API vectors derived from the training set are used for semantic distance calculation. An *extended feature set* is defined from the initial feature set after all new APIs are

included in feature extensions.

A toy example is shown in Fig. 3.5 to illustrate the feature extension. In Fig. 3.5, two clusters are formed in the vector space generated from a training set, including the one for *Feature P* containing API_A, API_B and API_C, and the one for *Feature Q* containing API_D, API_E and API_F.

Now consider a new API API_Y in the vector space generated from a testing set, an average semantic distance is calculated between API_Y and each cluster of *Feature P* and *Feature Q*. Since *Feature Q* has the least average distance from API_Y, it is expanded to include API_Y for *feature extension*. After that, *Feature Q* will contain four APIs: API_D, API_E, API_F and API_Y, while *Feature P* keeps unchanged. Note that API_B does not appear in testing set and thus has no representative API vector in the vector space, so it is simply excluded in the calculation.

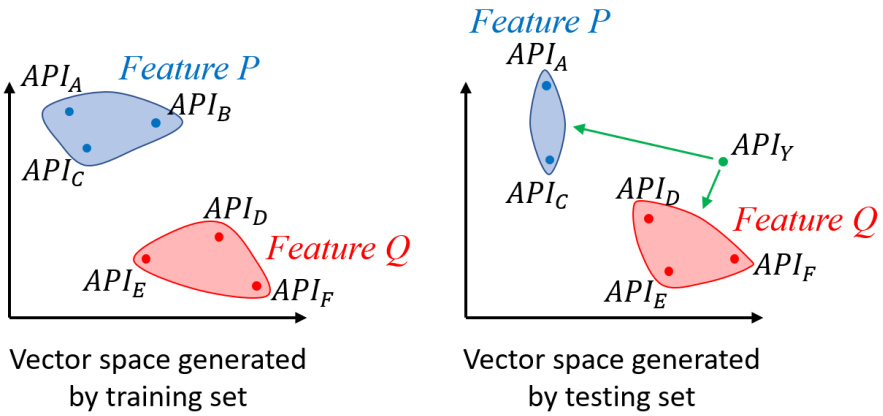


Figure 3.5: API Cluster Extension

The purpose of generating an extended feature set is to simulate new APIs' contributions to malware detection using existing APIs', where the former cannot be directly measured by a classification model due to the lack of true labels in testing sets.

3.2.4 Classification Model Training and Testing

Feature Vector Generation. Given a feature set, which is either initial feature set in the training phase, or extended feature set in the detection phase, SDAC generates a binary *feature vector* for each app by a one-to-one mapping from the feature set. An element in an app's feature vector is zero if none of the APIs in its mapped feature is used by the app, and it is one otherwise.

Classification Model Training. In the training phase, SDAC generates a feature vector for each app in the training set, and associates the feature vector with the app's true label. Then, SDAC trains a classification model with all feature vectors and associated labels.

Classification Model Testing. In the detection phase, SDAC generates a feature vector for each app in the testing set. Then, SDAC uses the trained classification model to output a predicted label for each app according to its feature vector as model input.

3.2.5 Model Voting

In SDAC, a feature in the feature set may consist of multiple APIs. It is possible that some of these APIs are used by a benign app, and some of them are used by a malicious app. A single feature can hardly be used to distinguish between benign and malicious apps. It is thus much better to leverage on all features in the feature set for malware detection. To further improves its accuracy, SDAC exploits the distinguishing power of multiple feature sets instead of a single feature set. In particular, SDAC performs its cluster algorithm for $m \geq 1$ times in its third step and thus generates m initial feature sets and m extended feature sets. A classification model is trained on each initial feature set in the training phase and then tested on the corresponding extended feature set in the detection phase. With total m classification models, SDAC outputs a predicted label "malware" for an app if at least $\tau \leq m$ out of m classification models

agree on such label, and it outputs "benignware" otherwise.

In our experiments, we discover that F-scores of SDAC increase by around 3% to 10% using multiple voting. The details on tuning τ and m are explained in section 3.4.

3.3 Two Modes of SDAC and Online Versions

While the basic SDAC focuses on processing one testing set only, it can be extended in two different modes, SDAC-FEO and SDAC-FMU, to process multiple testing sets $T_1, T_2, \ldots, T_N$ in which apps are developed in different time periods with some new APIs. The difference between SDAC-FEO and SDAC-FMU is that in the detection phase, SDAC-FEO performs feature extension only, while SDAC-FMU performs feature and model updates as well.

3.3.1 SDAC-FEO

In the training phase, SDAC-FEO takes a training set as input and outputs m initial feature sets and m classification models in the same way as the basic SDAC does. In the detection phase when it is applied to testing set T_1, each initial feature set is extended to an "extended feature set for T_1". Then, each app in T_1 is transformed to m feature vectors according to m "extended feature sets for T_1". Finally, the m classification models are used in model voting to predict a label for each app according to its feature vectors.

When SDAC-FEO is applied to testing set T_N ($N \geq 2$), each "extended feature set for T_{N-1}" is regarded as "initial feature set for T_N", and then it will be extended to an "extended feature set for T_N" in the same way as the "feature extension" step in the basic SDAC. After that, each app in T_N is transformed to m feature vectors according to m "extended feature sets for T_N". The same m classification models are used as before to predict a label for each app according to its feature vectors.

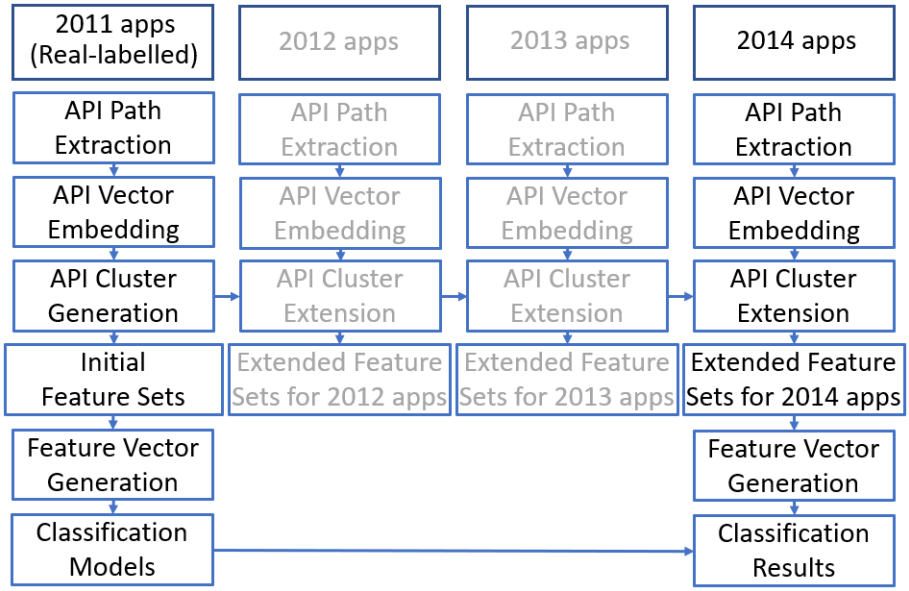


Figure 3.6: Structure of SDAC-FEO with One Training Set (2011 apps) and Three Testing Sets (2012 apps, 2013 apps, and 2014 apps)

Fig. 3.6 shows the structure of SDAC-FEO with one training set (2011 apps), and three testing sets (2012 apps, 2013 apps, and 2014 apps). In this figure, the set of 2011 apps is used to generate initial feature sets and train classification models. The same classification models are used to classify 2014 apps after they are applied to 2012 apps and 2013 apps. The extended feature sets for 2014 apps are generated by extending the initial feature sets for 2011 apps three times in a sequence.

3.3.2 SDAC-FMU

SDAC-FMU is the same as SDAC-FEO in its training phase, and in the detection phase when it is applied to testing set T_1. It performs additional steps on feature and model updates when it is applied to other testing sets $T_2, \ldots, T_N$.

Now assume that SDAC-FMU has been applied to testing sets $T_1, \ldots, T_{N-1}$, producing a prediction label (i.e., pseudo-label) for each app in these sets. When SDAC-FMU is applied to testing set T_N, it first generates m "initial feature sets for T_N" (in the same way as the basic SDAC generates initial feature sets) from

the union of the training set and $T_1, \ldots, T_{N-1}$. We call this process *feature update*. Note that this is different from SDAC-FEO where the initial feature sets never change.

Then, SDAC-FMU trains m classification models for T_N from scratch using (i) the apps in the training set with their true labels, and (ii) the apps in $T_1, \ldots, T_{N-1}$ with their pseudo-labels, in which each app is converted to m feature vectors according to m "initial feature sets for T_N". We call this process *model update*. Note that no true labels are available for the apps in testing sets in our assumption; thus, SDAC-FMU uses pseudo-labels for the apps in $T_1, \ldots, T_{N-1}$ for model update.

After model update, SDAC-FMU extends each "initial feature set for T_N" to an "extended feature set for T_N" (in the same way as the basic SDAC extends an initial feature set). Then, each app in T_N is converted to m feature vectors according to m "extended feature sets for T_N". Finally, the classification models for T_N which have been trained in model update are used to predict a label for each app in T_N according to its feature vectors.

Fig. 3.7 shows the structure of SDAC-FMU with one training set (2011 apps), and three testing sets (2012 apps, 2013 apps, and 2014 apps). When SDAC-FMU is applied to 2014 apps after it is trained with 2011 apps and applied to 2012 apps and 2013 apps, it first generates initial feature sets for 2014 apps from the union of 2011 apps, 2012 apps, and 2013 apps. Then, SDAC-FMU generates feature vectors for each app in the union, and trains classification models for 2014 apps using (i) 2011 apps with their true labels, and (ii) 2012 and 2013 apps with their pseudo-labels, where each app is converted to its feature vectors for model training. Finally, SDAC-FMU extends the initial feature sets with 2014 apps, converts each 2014 app to feature vectors according to the extended feature sets, and uses the classification models for 2014 apps to classify each 2014 app by its feature vectors.

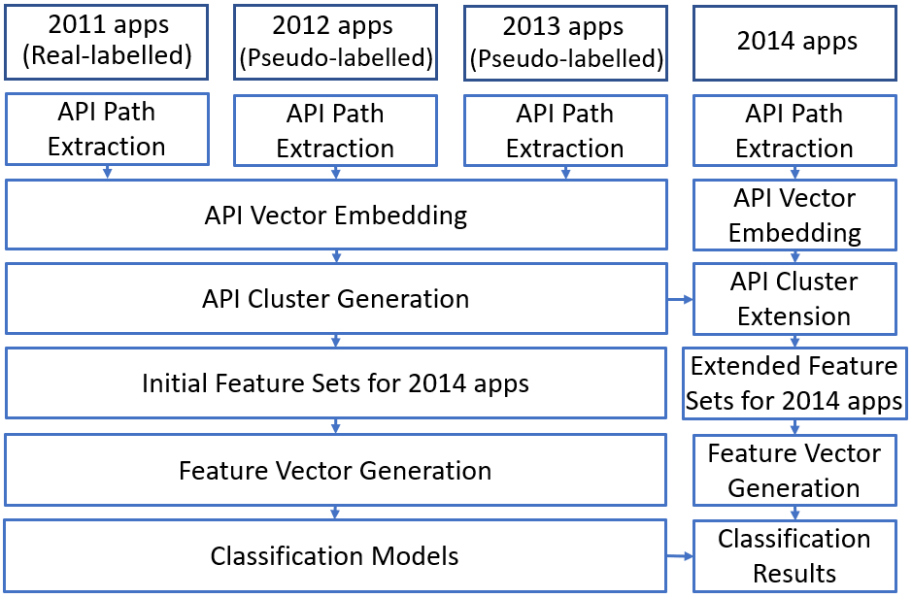


Figure 3.7: Structure of SDAC-FMU with One Training Set (2011 apps) and Three Testing Sets (2012 apps, 2013 apps, and 2014 apps)

3.3.3 Online Versions

Both SDAC-FEO and SDAC-FMU require that the current testing set T_N be available for performing feature extension before they can be applied to classify each individual app in this testing set. To overcome this restriction, we develop their online versions, SDAC-FEO-OL and SDAC-FMU-OL, in which the feature extension step is skipped, for classifying individual apps in time without waiting for the whole testing set to be available.

SDAC-FEO-OL. SDAC-FEO-OL is the same as SDAC-FEO in the training phase, which generates m initial feature sets and m classification models. When SDAC-FEO-OL is used to classify each app in T_1, it converts the app to m feature vectors according to m *initial feature sets*. Then, it uses m classification models that have been trained to output a predicted label for each app according its feature vectors. After all apps in T_1 have been classified, SDAC-FEO-OL generates m extended feature sets for T_1 the same way as SDAC-FEO does from the whole set T_1.

When SDAC-FEO-OL is used to classify each app in T_N ($N \geq 2$), it

converts the app to m feature vectors according to m *extended feature sets for* T_{N-1}. Then, it uses the same classification models that have been trained to output a predicted label for each app in T_N via its feature vectors.

After all apps in T_N have been processed, SDAC-FEO-OL resorts to SDAC-FEO to process T_N again, generating m extended feature sets for T_N. This is to prepare SDAC-FEO-OL for detecting apps in the next time period.

SDAC-FMU-OL. SDAC-FMU-OL is the same as SDAC-FMU in the training phase, which generates m initial feature sets for T_1 and m classification models for T_1. SDAC-FMU-OL is the same as SDAC-FEO-OL when it is applied to classify each app in T_1 according to m initial feature sets for T_1 using m classification models for T_1.

With all apps in T_1 being processed, SDAC-FMU-OL resorts to SDAC-FMU to process T_1 again, generating m initial feature sets for T_2, and m classification models for T_2.

When SDAC-FMU-OL is applied on testing set T_N ($N \geq 2$), it converts each app to m feature vectors according to m initial feature sets for T_N. Then, it uses the m classification models for T_N to predict label for each app in T_N according its feature vectors. After all apps in T_N are processed, SDAC-FMU-OL performs feature and model updates the same way as SDAC-FMU does with the whole set T_N.

Notes. When an individual app is detected online, the app is first converted to m feature vectors according to *the set of APIs* it used, and then classified by m classification models. The first three steps of SDAC (API path extraction, API vector embedding, and API cluster generation and extension) are not performed in this process, which makes the online versions much faster than the offline ones.

SDAC-FEO-OL and SDAC-FMU-OL are different from direct applications of online machine learning in malware detection [59] since our online versions do not require true labels to be used for model updates, while online machine

learning does require [86].

3.4 Evaluation of SDAC

Dataset. SDAC is evaluated using a dataset of around 36k benignware samples and 35k malware samples randomly chosen from an open Android application collection project [16]. Table 3.1 shows an overview of our dataset, which consists of benignware samples and malware samples developed in six years from 2011 to 2016. The time of each app is defined as the time its APK file was packaged, which can be found in the .dex file inside its APK [61].

The labels of the samples in our dataset were decided according to the reports from VirusTotal [10] which we obtained in July 2018. Based on the reports, we labelled apps with zero positive result as "benign", and apps whose reports containing more than a threshold T_{mal} positive results as "malicious".

In the literature of malware detection, different values of T_{mal} are used for labelling "malicious" apps. According to Roy et al., malware samples that received one positive report only from VirusTotal were considered to be of "low quality," and those received more than ten positives out of 54 scanners were considered "high quality" [69]. Arp et al. labelled an app as malicious if it received at least 20% positive results from a set of selected scanners [19]. Alex et al. performed a large-scale study on aggregating the results of scanners from Virustotal and deemed a malicious label to be trustful if it came from 4 or more positive scanning results out of 34 different scanners [43].

In our experiments, we evaluate SDAC on 3 different datasets labelled with $T_{mal} = 4, 9\ and\ 15$, respectively, out of total 63 scanners in Virustotal[2]. We set $T_{mal} = 15$ in the default case and show our results for $T_{mal} = 4$ and $T_{mal} = 9$ in section 4.5.

[2]Lists of .apk file hashes in these datasets can be found at http://dx.doi.org/10.21227/rasc-k457.

Table 3.1: Overview of Dataset($T_{mal} = 15$)

Year	2011	2012	2013	2014	2015	2016
Benign	6072	5887	5920	5934	5929	5903
Malware	4961	5953	5877	5902	5925	5879
# total APIs	14842	16213	17519	17714	17933	19933
# new APIs	n/a	3369	3407	2701	2902	4009
% new APIs	n/a	20.78	19.45	15.645	16.18	20.11

Table 3.1 shows the number of unique new APIs and the number of all APIs in different years from the dataset labelled with $T_{mal} = 15$. An API is considered to be new in a year if it is not used by any app that was developed before that year in our dataset.

Tools and Parameters. We choose the following tools and parameters for the evaluation of SDAC. In the API path extraction step, we choose FlowDroid to extract a directed call graph among program classes from each app [20]. The parameter d used for API path extraction is chosen to be the same as the window size for API vector embedding. In the API vector embedding step, we rely on the gensim toolkit [67] to implement the Skip-Gram model and derive API vectors from a set of apps, where we choose window size $S = 5$ and API vector size $K = 200$ (i.e., the dimension of API vector space).

In the API cluster generation and extension step, we choose the same-size k-means cluster algorithm from open-source data mining framework ELKI [35], and set the number of clusters $k = 1000$. In the classification model training and testing step, we choose linear SVM models as our classification models, and set the number of classification models $m = 9$, and the threshold $\tau = 3$ in model voting.

We tune these parameters, as well as other parameters (e.g., iteration times and learning rate for API vector embedding), to produce the best results when SDAC is trained and tested in cross validation using the same training set (2011 apps) under the constraint of our computing resources (a desktop computer

with 3.3 GHz CPU and 12GB memory). These parameters are used across all experiments for the evaluation of SDAC on various testing sets.

Selection of Parameters: k, m and τ. The parameters k (as in k-means clustering algorithm), m (i.e. the number of classification models used by SDAC) and τ (i.e. the threshold used in model voting) are tuned for the best performance of SDAC in the cross-validation on the training set, which is the 2011 dataset in our experiments.

Fig. 3.8 shows the performance of SDAC in cross-validation with different k values. The F-score of SDAC increases rapidly from $k = 50$ to $k = 500$, and remains stable after $k = 1000$. Since a higher k costs more time on model training and testing, we choose $k = 1000$ in our experiments.

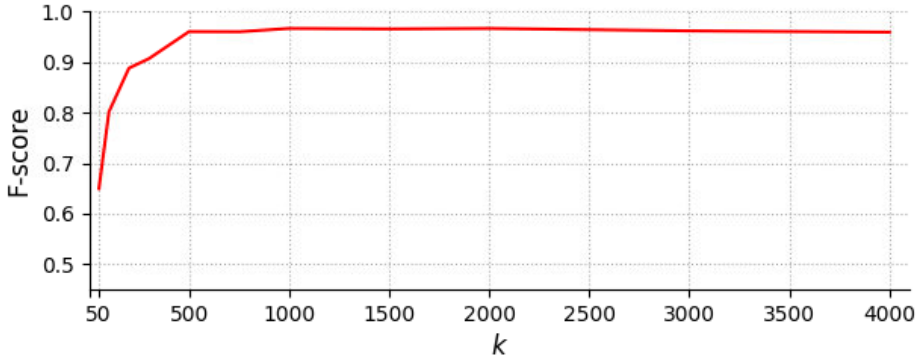


Figure 3.8: F-score of SDAC in Cross Validation on 2011 Training Set with Different k) and Three Testing Sets (2012 apps, 2013 apps, and 2014 apps)

Fig. 3.9 shows how the numbers m of classification models and threshold τ are decided. In the cross-validation experiments, SDAC reaches its highest F-score (above 97%) when m>7 and τ/m is around 30% to 40%. Since the overhead of SDAC is proportional to m, we choose $m = 9$ and $\tau = 3$, which reaches the highest F-score with the smallest m.

3.4.1 Evaluation of SDAC-FEO

Three sets of experiments are conducted to evaluate SDAC-FEO. The first set is conducted to evaluate the accuracy of SDAC-FEO when it is trained and tested

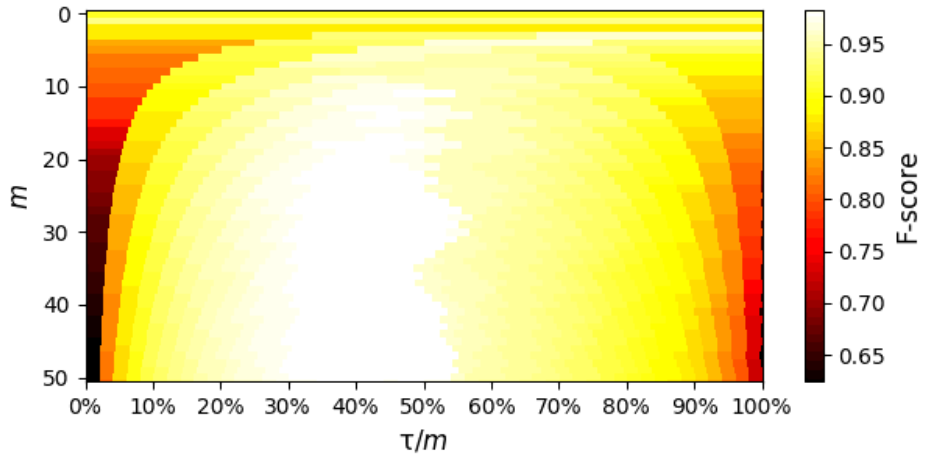


Figure 3.9: F-score of SDAC in Cross Validation on 2011 Training Set with Different m and τ

in cross validation using a set of samples developed in the same time period. The second set of experiments is conducted to evaluate the aging speed of SDAC-FEO when it is trained on a set of samples developed in one time period, and tested on other sets of samples developed in later time periods. The third set of experiments is conducted to compare the accuracy and aging speed of SDAC-FEO with MaMaDroid [53].

The accuracy of a malware detection model can be measured in F-score on a set of malware and on a set of benignware. F-score is the harmonic mean of $precision$ and $recall$, where $precision = |TP|/(|TP| + |FP|)$ and $recall = |TP|/(|TP| + |FN|)$. We use TP (i.e., true positives) to denote the set of malware that is correctly detected as malware, FP (i.e., false positives) the set of benignware that is incorrectly detected as malware, FN (i.e., false negatives) the set of malware that is incorrectly detected as benignware, and TN (i.e., true negatives) the set of benignware that is correctly detected as benign.

The accuracy of SDAC-FEO is first evaluated in 5-fold cross validation using samples developed in the same time period. Table 3.2 shows the accuracy of SDAC-FEO in terms of precision, recall, and F-score on different set of apps, which is denoted by the time period in which the apps were developed. The average F-score of SDAC-FEO is 98.25%, which serves as a good starting point

Table 3.2: F-score of SDAC-FEO in Cross Validation

App set	Precision	Recall	F-Score
2011	0.9885	0.9672	0.9778
2011~12	0.9918	0.9842	0.9880
2011~13	0.9906	0.9848	0.9877
2011~14	0.9915	0.9829	0.9872
2011~15	0.9905	0.9812	0.9858

for evaluating SDAC-FEO over time.

SDAC-FEO Performance Over Time. The aging property of SDAC-FEO is evaluated in a series of experiments in which SDAC-FEO is trained on a set of samples developed in one time period and tested on other sets of samples developed in later time periods. Fig. 3.11 shows the F-score of SDAC-FEO in detection over time. When SDAC-FEO is evaluated on a testing set that is newer than the training set by one year, its average F-score is 97.49%, which declines by 1.03% from its average F-score in cross validation (98.52%). It declines further to 95.02%, 88.48%, 78.22%, 73.72% when SDAC-FEO is evaluated on testing sets that are newer than the training sets by two, three, four, and five years, respectively. The average aging speed of SDAC-FEO is 4.96% in F-score per year in these experiments.

Analysis on API Cluster Extension. API cluster extension is a critical step in SDAC. In this step initial feature sets are extended with new APIs to create the extended feature set. This enables SDAC to evaluate new APIs' contributions to malware detection with the existing classification models, which have been trained by a set of labelled apps in which none of the new APIs are used.

To further understand how API cluster extension contributes to slow aging of SDAC-FEO, in this section, we calculate the changes on feature vectors of testing set apps with and without feature extension, and then extract each feature's weight in the linear SVM model which helps to figure out how such change will affect the detection results.

In detail, in the SVM model used by SDAC-FEO, the inner product between an app's feature vector and the linear SVM model's weight vector is the *output score* of the app's feature vector in the SVM model [19], which represents the confidence of the SVM model in classifying the app as either malware if the output score is positive, or benignware if the output score is negative. The confidence of an SVM model is proportional to the absolute value of an output score.

For each app in a testing set, we transform it to two feature vectors according to an initial feature set and an extended feature set for each SVM model, respectively. The *output score difference* of an app is defined as the output score of its feature vector derived from the extended feature set subtracted by the output score of its feature vector derived from the initial feature set. The output score difference of an app in an SVM model represents the change in confidence on the app caused by API cluster extension. A positive (negative, respectively) output score difference means more confidence in classifying an app as malware (benignware, respectively).

Then, we examine the average output score difference for all malware samples in a testing set and for all classification models used by SDAC-FEO. We also examine the average output score difference for all benignware samples in the same testing set. The *distinguishability of API cluster extension* in each experiment (with a training set and a testing set) is defined as the average output score difference for all malware samples subtracted by the average output score difference for all benignware samples. If the distinguishability of API cluster extension is positive, then the API cluster extension makes a positive contribution to malware detection, and thus contributes to slow aging of SDAC-FEO.

Fig. 3.10 shows the *distinguishability* brought by API cluster extension in our experiments. The contributions of API cluster extension to malware detection are positive in all of our experiments, which demonstrates that the

feature extension will indeed contribute to the accuracy of SDAC.

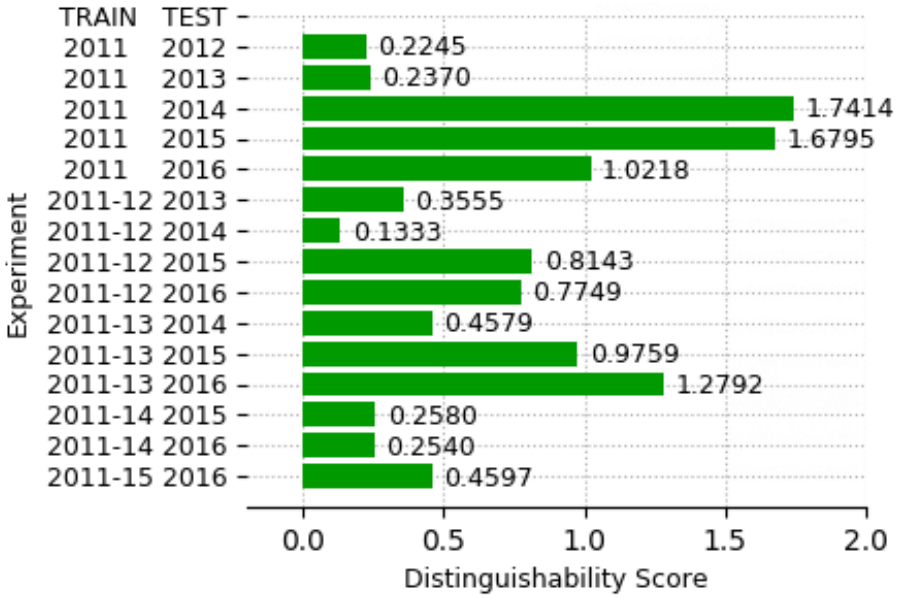


Figure 3.10: Distinguishability of API Cluster Extension

Aging Slower Than MaMaDroid. The aging speed of SDAC-FEO is compared with MaMaDroid using the same dataset. MaMaDroid is the only solution which we know to be resilient to the changes in Android specifications, and has a significant better performance than other solutions such as DroidAPIMiner [53]. In particular, MaMaDroid first derives API paths from each app using FlowDroid, and abstracts APIs to their corresponding packages (or package families) in the API paths. It then summarizes all abstracted paths to a Markov model, and converts the Markov model to a feature vector for each app, where each feature in the feature vector represents a transition between two existing packages in the Markov model. After that, it trains a machine learning classification model from a training set of apps according to their feature vectors and associated true labels. By abstracting APIs to packages, MaMaDroid is resilient to the adding of new APIs to existing packages in Android specifications; however, it is not designed to be resilient to the adding of new API packages.

We re-implemented MaMaDroid using its sourcecode [52]. Both SDAC-

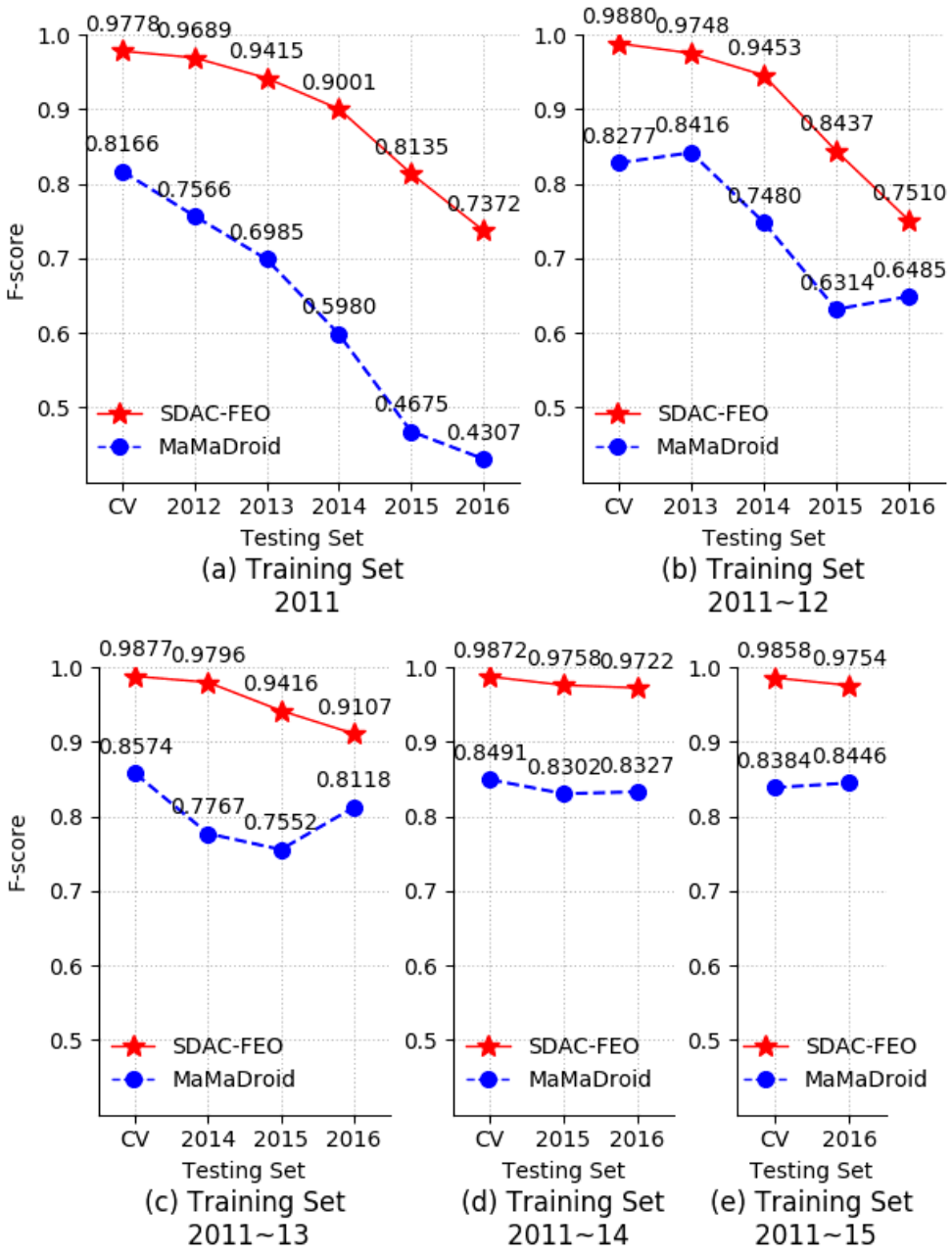


Figure 3.11: Comparison between SDAC-FEO and MaMaDroid (CV: 5-fold cross validation)

FEO and MaMaDroid rely on FlowDroid to decompile app Apk files; they were evaluated using the same training sets, testing sets, standard SVM classification models, and F-score measurement.

We note that in the original MaMaDroid paper [53], the random forests algorithm produces the best malware detection results among four classification algorithms, including random forests, 1-NN, 3-NN, and SVM. However, in our

experiments, MaMaDroid performs the best with SVM.

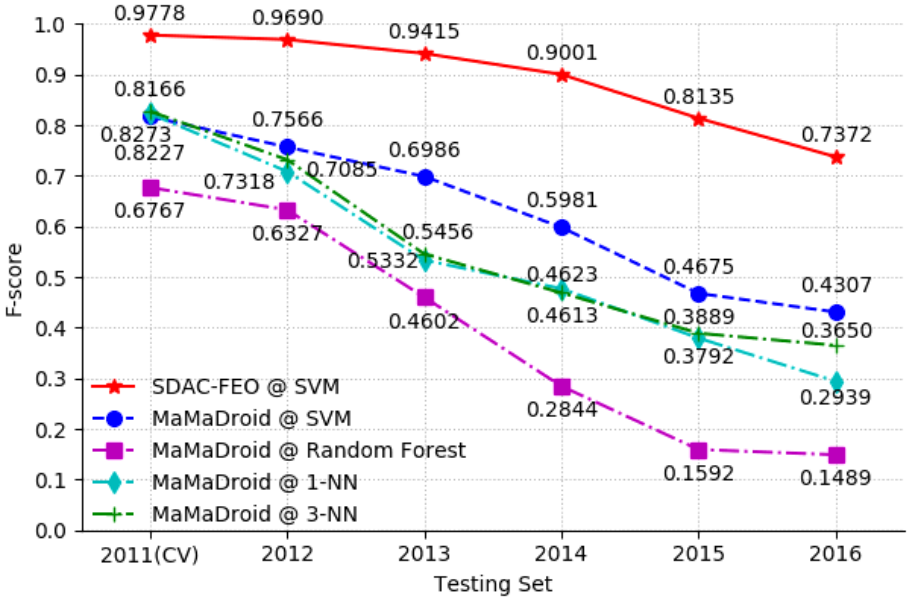


Figure 3.12: Evaluation of MaMaDroid with Different Classification Algorithms (CV:5-fold cross validation)

In detail, we re-implement MaMaDroid with all four classification algorithms, including random forests, 1-NN, 3-NN, and SVM as mentioned in its original paper [53], and test the performance of MaMaDroid with each algorithm in the default setting of our experiments. Fig. 3.12 shows that the performance of MaMaDroid with SVM outperforms 1-NN, 3-NN and random forests consistently from 2011 to 2016. Therefore, we choose SVM as the classification model for MaMaDroid in our experiments.

Fig. 3.11 shows that the performance of SDAC-FEO is significantly and consistently better than MaMaDroid in all experiments. In particular, the average F-scores of SDAC-FEO when it is evaluated on testing sets that are newer than training sets by one to five years are 97.49%, 95.02%, 88.48%, 78.22%, and 73.72%, respectively. In comparison, the average F-scores of MaMaDroid in the corresponding cases are 81.00%, 75.86%, 68.05%, 55.80%, and 43.07%, respectively. The average F-score of SDAC-FEO is higher than MaMaDroid by 20.65% when they are evaluated on the same training set and testing set across all experiments. In terms of aging speed, SDAC-FEO declines

by 4.96% in F-score per year on average over five years, while MaMadroid declines by 8.15% in the same case.

We notice that the performance of MaMaDroid in our experiments is not as good as what was reported in [53]. One possible reason is that overlaps exist between the training sets and the testing sets used in [53], where two benign sets were collected from PlayDrone [79] and Google Play store, and five malware sets were collected from Drebin [19] and VirusShare [11]. For one set of experiments in [53], the same "oldbenign" benign set was used to mix with various malware sets dated from 2012 to 2016 to form training sets and testing sets. For another set of experiments, the same "newbenign" benign set was used to form all training sets and testing sets. Such overlaps made it difficult to evaluate how MaMaDroid aged over time.

3.4.2 Evaluation of SDAC-FMU

Aging Slower Than SDAC-FEO and MaMaDroid. Compared to SDAC-FEO, SDAC-FMU takes additional steps of feature update and model update for better performance. Fig. 3.13 shows that its performance is significantly and consistently better than SDAC-FEO. The average F-score of SDAC-FMU (97.39%) is higher than SDAC-FEO (92.89%) by 4.50% when they are evaluated on the same training set and testing set across all experiments. In terms of aging speed, SDAC-FMU declines by 0.25% in F-score per year on average over five years, while SDAC-FEO declines by 4.96% in the same case.

Since SDAC-FMU updates its classification models with pseudo-labels, we also update MaMaDroid classification models with pseudo-labels for a fair comparison. Fig. 3.13 also shows that the performance of MaMaDroid updated with pseudo-labels is almost the same as before. One possible reason is that MaMaDroid neglects the APIs of new packages in generating these pseudo-labels, and the updating will then reinforce the mistakes caused by neglecting

these new APIs in MaMaDroid's model.

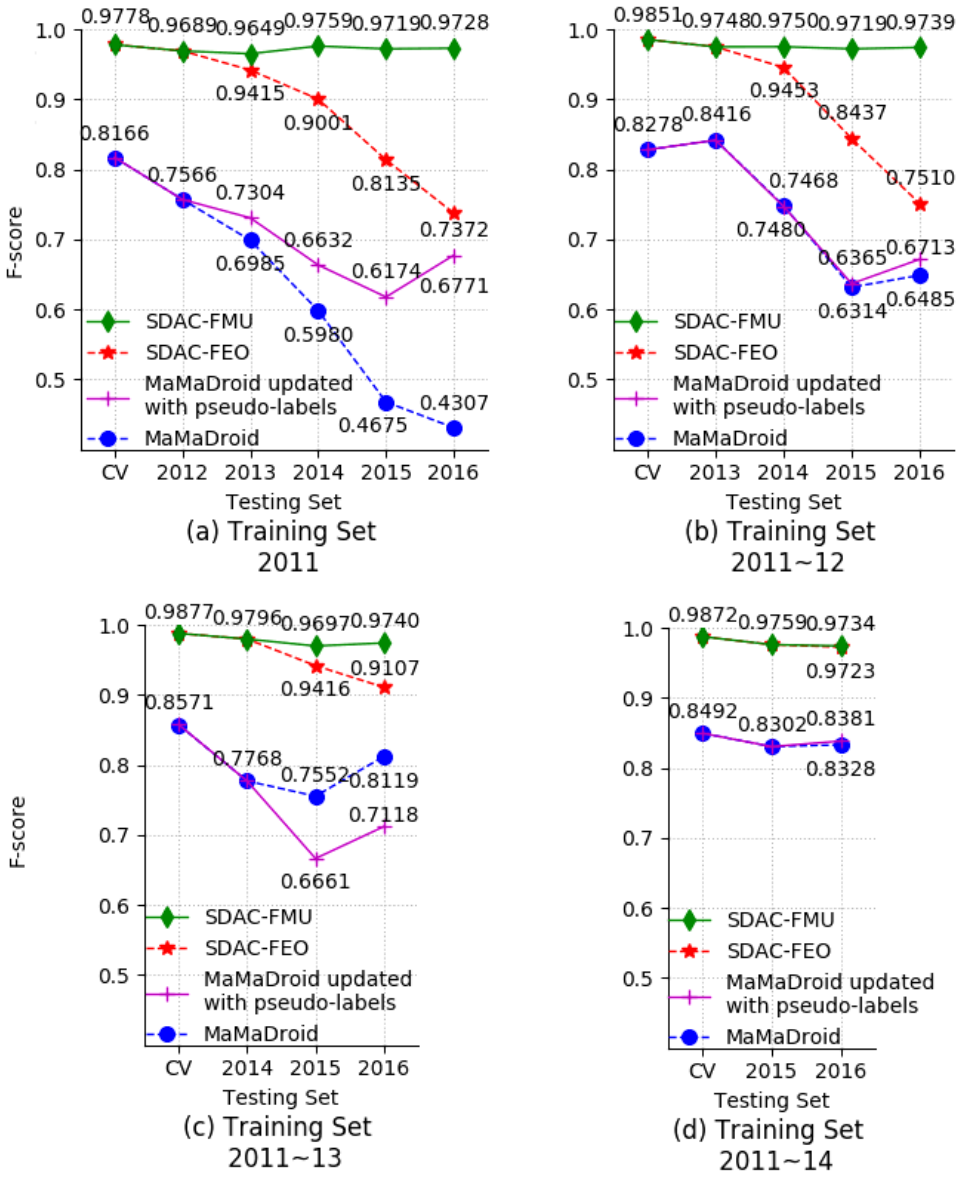


Figure 3.13: Comparison between SDAC-FMU, SDAC-FEO, and MaMaDroid (CV: 5-fold cross validation)

False Positives. We examine the misclassified results of SDAC-FMU when it is trained on 2011 set and evaluated on five testing sets, dated from 2012 to 2016. To understand why false positives are misclassified, we compute a *weight for an API* by averaging the weights of the features that contain this API in all SVM models when SDAC-FMU is applied to each testing set. The weight of an API can be used to measure its contribution to the confidence of SDAC-FMU in

classifying an app. We sort all APIs according to their weights for each testing set. We choose the top p and the bottom p APIs in each sorted list, which are the APIs with most contributions to the confidence of SDAC-FMU in classifying an app as malware and as benignware, respectively.

For a set of apps, we define *API ratio for an API* as the percentage of the apps in the set that use this API. We further define *top-p ratio* (*bottom-p ratio*, respectively) as the average of API ratios for the top p APIs (bottom p APIs, respectively). The top-p ratio subtracted by the bottom-p ratio for a set of apps means the confidence of SDAC-FMU in classifying the set as malware.

Fig. 3.14 shows typical values of top-p ratio subtracted by bottom-p ratio for true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN), where $p = 1\%$. Compared to TN, SDAC has more confidence in classifying FP as malware, and the most confidence in classifying TP as malware.

In the list of top-weighted APIs, we discovered some APIs such as *getConfiguration()* and *getDeviceId()* that were considered as "dangerous" in previous research [19, 53]. Benign apps using such APIs are more likely to be detected as malware by SDAC. For example, 40% (126/315) of false positives and more than a half of true positives (3310/5890) use API *getConfiguration()*, while only 23% (1289/5555) of true negatives makes use of it. For another example, 66% (208/315) of false positives and 96% (5633/5890) of true positives include API *getDeviceId()* in class *TelephonyManager*, while only 22% (1245/5555) of true negatives use it.

False Negatives. We also examine the false negatives of SDAC-FMU when it is trained on 2011 set and evaluated on five testing sets, dated from 2012 to 2016. Among 1076 false negative samples generated from all testing sets, about 69% (638/931) of them are classified as "positive: adware" by at least one VirusTotal scanner. According to TrendMicro [75], the adware apps may come from repackaging benign apps with 3rd party advertisement libraries; it is

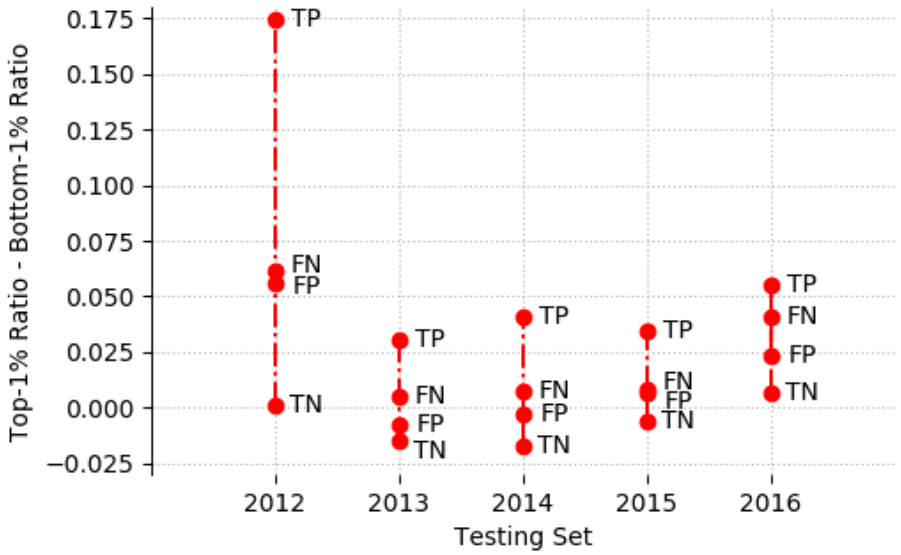


Figure 3.14: Difference in API Usage among TP, FP, FN, and TN with 2011 Training Set

difficult for SDAC-FMU to distinguish them from true benign apps.

Besides, about 8.5% (79/931) false negative samples are classified as "positive: riskware", and about 17.4% (162/931) as "positive: not-a-virus" by at least one VirusTotal scanner. According to the explanation from Kaspersky Lab [44], riskware refers to legtimate programs which are easy to be exploited by malicious attackers, and not-a-virus is associated with adware and riskware.

Fig. 3.14 also shows that compared to TP, SDAC has more confidence in classifying FN as benignware, and the most confidence in classifying TN as benignware.

3.4.3 Evaluation of SDAC-FEO-OL & SDAC-FMU-OL

The performances of SDAC-FEO-OL and SDAC-FMU-OL are evaluated in the default case, which is formed with the smallest training set (2011 apps) and the longest time span across testing sets (2012-2016) in our experiments. Fig. 3.15 shows that SDAC-FMU-OL performs very closely to SDAC-FMU, while the performance gap between SDAC-FEO-OL and SDAC-FEO is more obvious. Compared to SDAC-FMU, the F-score of SDAC-FMU-OL declines by 0.41%

on average, by -0.29% in minimum (for 2014 testing set), and by 0.85% in maximum (for 2012 testing set). Compared to SDAC-FEO, the F-score of SDAC-FEO-OL declines by 3.45% on average, by -0.30% in minimum (for 2013 testing set), and by 7.97% in maximum (for 2014 testing set). Nonetheless, both SDAC-FMU-OL and SDAC-FEO-OL perform significantly better than MaMaDroid.

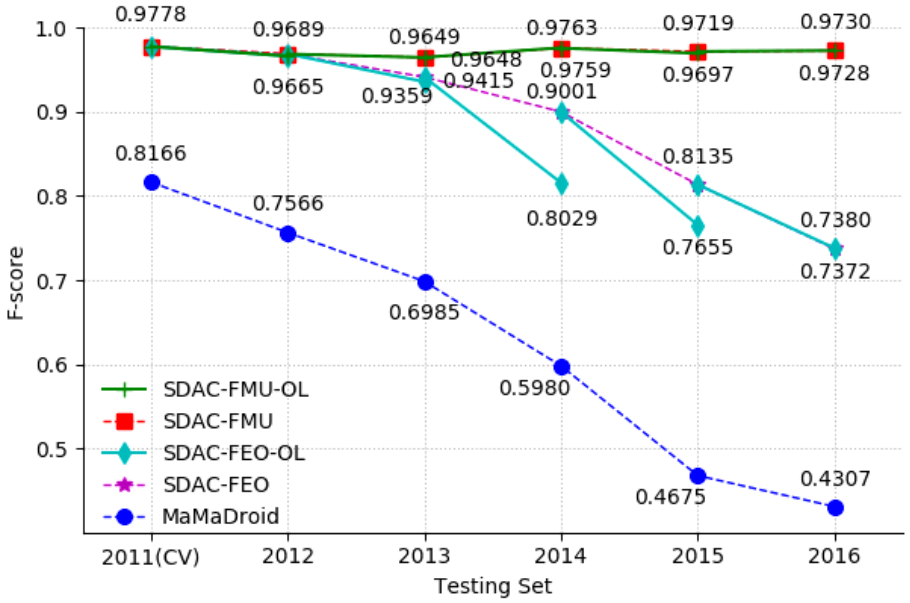


Figure 3.15: Evaluation of Online Versions with 2011 Training Set (CV: 5-fold cross validation)

3.4.4 Runtime Performance

The runtime performance of SDAC is evaluated on a desktop computer using one Intel(R) i5-4590 3.3 GHz CPU and 12 GB physical memory running on the Ubuntu 14.04 (LTS) operating system. Table 3.3 shows the runtime performance of SDAC in all four steps: (i) API Path Extraction, (ii) API Vector Embedding, (iii) API Cluster Generation and Extension, and (iv) Classification Model Training and Testing.

Runtime of SDAC-FEO-OL and SDAC-FMU-OL in Detection Phase. SDAC-FEO-OL can be used to detect individual apps online without waiting for

Table 3.3: Runtime Performance of SDAC

Step		Runtime
API Path Extraction	Call Graph Generation	Avg. 37.29 sec. per app (min. 3.12s & max. 1184.33s)
	API Path Extraction	Avg. 16.82 sec. per app (min. 8.30e-4s & max. 1350.06s)
API Vector Embedding	Transform APIs into Vectors	333.20 sec. (11033 apps) ~ 3154.48 sec. (58360 apps)
API Cluster Generation and Extension	API Cluster Generation (in training phase)	85.91 sec. (11033 apps) ~ 300.33 sec. (58360 apps)
	API Cluster Extraction (in testing phase)	48.93 sec. (11033 apps) ~ 25.77 sec. (58360 apps)
Classification Model Training and Testing	Classification Model Training (in training phase)	9.77 seconds (11033 apps) ~ 1683.75 seconds (58360 apps)
	Classification Model Testing (in testing phase)	Avg. 8.90e-4 sec. per app (min. 4.57e-4s & max. 1.59e-3s)

a whole testing set to be available. The time cost for detecting an app online is 0.20 seconds on average. Once a whole testing set is available, SDAC-FEO-OL extends its feature sets using the whole testing set in the same way as SDAC-FEO does. This additional time cost is similar to SDAC-FEO in its detection phase.

The time cost of SDAC-FMU-OL is the same as SDAC-FEO-OL for detecting an app online. Once a whole testing set is available, SDAC-FMU-OL performs feature and model updates in the same way as SDAC-FMU, so the additional time cost is also same as in SDAC-FMU.

Runtime of MaMaDroid. MaMaDroid takes three major steps in both training phase and detection phase: (i) FlowDroid is exploited to derive a set of API paths from an app, (ii) a Markov model is formed from a set of API paths, and then used to compose a feature vector, and (iii) a classification model is trained from (in training phase) or applied to (in detection phase) a set of apps. In our implementation, the training phase of MaMaDroid takes 37.29 seconds on average in step one, 0.41 seconds on average in step two, and 16.3 seconds (785.29 seconds, respectively) for processing a set of 11,033 apps (58,360 apps, respectively) in step three. In its detection phase, MaMaDroid takes 0.0036

seconds on average for classifying a single app in step three, while the first two steps take the same time as in the training phase.

Runtime Performance Comparison. In the training phase, SDAC spends more time (54.17 seconds per app on average) than MaMaDroid (37.70 seconds per app on average) for transforming decompiled codes into feature vectors. In the next step of classification model training, which refers to model training in SDAC-FEO or model updating in SDAC-FMU, the time cost ranges from 9.77 seconds (on the smallest training set containing 11033 apps) to about 0.47 hours (on the largest training set containing 58033 apps) for training each classification model. Since 9 classification models are used in SDAC, the total time cost of SDAC for this step ranges from 87.93 seconds to about 4.2 hours. In comparison, MaMaDroid spends 16.3 seconds to about 13.1 minutes on different training sets in the model training step. In the detection phase, the average time for detecting each app is 8.90e-4 seconds for SDAC, and 3.6e-3 seconds for MaMaDroid.

Although SDAC takes longer training time than MaMaDroid, it achieves much higher accuracy and slower aging speed as shown in sections 4.1 to 4.3. The runtime performance of SDAC is acceptable in all out experiments even though they are conducted on a common desktop computers (i.e., i5-4590@3.3GHz CPU and 12GB memory).

Notes. For both SDAC-FMU and SDAC-FMU-OL, the time cost for feature and model updates increases with the size of its input data, which is the union of its training set and all past testing sets. The size of the input data keeps increasing as more testing sets are processed and accumulated over time. To address this problem, we suggest to apply a validation window to the input data which covers all past testing sets starting from the last testing set in which all apps' true labels are available[3]. The size of this validation window is limited, and so is the time

[3]Relaxing our assumption, we believe that true labels of testing apps will be finally available after a limited period of time.

cost for feature and model updates.

The time granularity in forming testing sets is mainly decided by the number of apps that were collected within each time granularity, and each testing set should be large enough to extract accurate API context information from it. We suggest to choose each testing set to be larger than 5500 apps based on our experience with SDAC.

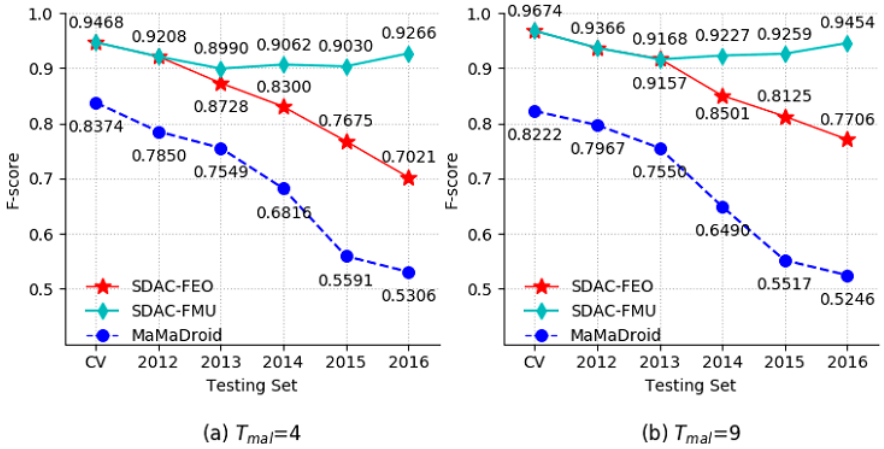
3.4.5 Evaluation of SDAC with Different T_{mal}

The performance of SDAC-FEO and SDAC-FMU are also evaluated on datasets that are labelled with different positive threshold $T_{mal} = 4$ and $T_{mal} = 9$ in VirusTotal reports. We also run MaMaDroid on these datasets for performance comparison.

Fig. 3.16 shows the F-measurements of SDAC when $T_{mal} = 4$ and $T_{mal} = 9$, respectively. In both cases, the 2011 app set is used as the training set and the 2012˜2016 data sets are used as the testing sets. In the case of $T_{mal} = 4$, SDAC-FEO declines by 4.89% in F-score and SDAC-FMU declines by 0.40% per year, while MaMaDroid declines by 6.12% per year on average. The advantages of SDAC-FEO and SDAC-FMU over MaMaDroid are 14.85% and 22.55%, respectively, in F-score on average. When $T_{mal} = 9$, the average aging speed in F-score is 3.94% for SDAC-FEO, 0.44% for SDAC-FMU, and 5.95% for MaMaDroid. The advantages of SDAC-FEO and SDAC-FMU over MaMaDroid are 19.22% and 25.22%, respectively, in F-score on average.

3.4.6 Evaluation of SDAC with Unbalanced Datasets

On our balanced datasets, SDAC outperforms MaMaDroid significantly. However, according to a recent research project, a balanced dataset may lead to biased results in malware detection since malware is usually the minority class (as compared to benignware) in the wild. It was reported that the ratio



(a) T_{mal}=4 (b) T_{mal}=9

Figure 3.16: Comparison between SDAC-FMU, SDAC-FEO, and MaMaDroid with $T_{mal} = 4$ and $T_{mal} = 9$ (CV: 5-fold cross validation)

of malware is around 10% in a real-world setting [64]. To estimate SDAC's performances in this case, we downsample malware to make its ratio to be 10% out of all apps, and form new datasets in our evaluation. The performances of SDAC on such datasets are shown in figure 3.17, which demonstrate wider differences between SDAC and MaMaDroid, and a similar trend as shown in our previous evaluations.

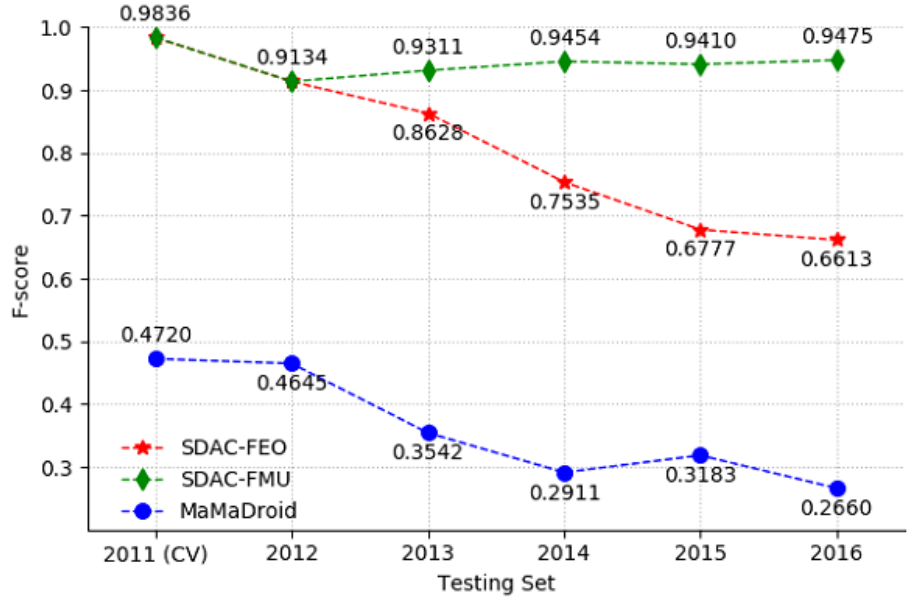


Figure 3.17: Comparison between SDAC-FMU, SDAC-FEO and MaMaDroid with Datasets of Unbalanced Ratio (CV: 5-fold cross validation)

3.5 Discussions

3.5.1 SDAC against Obfuscation

Code obfuscation tools, such as DroidChameleon [66] or [2] are often used to obfuscate malicious apps to avoid detection. Since SDAC presents its detection based on Android APIs, the obfuscation methods can be mainly classified into three categories by their impacts on the APIs used in apps: category-I: the set of APIs used by an app is not changed in obfuscation (e.g. encrypt native exploit or payload, rename identifier, identifiers or package), category-II: the set of APIs used by an app is enlarged to include new APIs in obfuscation (e.g. repackage, or insert junk code) and category-III: the set of APIs used by an app is reduced during obfuscation (e.g. method call hiding or reflection[17, 51]).

SDAC is naturally robust to category-I obfuscation methods since it detects an app solely based on the set of APIs used by it. Category-II obfuscation methods, such as "insert junk code" and "repackage malware into benignware", may enlarge the set of APIs used by an app, and thus avoid detection by SDAC. To test the robustness of SDAC against category-II methods, we generate a collection of 10,000 API sets for each testing set. Each API set in that collection is the union of the API set used by a benign sample and that by a true-positive malware sample randomly chosen from the testing set. One united API set is considered to be derived from a virtual malicious app created by "injecting" the code of a malware sample into the code of a benign sample. Then SDAC is applied to such virtual malicious apps to check its recall rates in various modes on different testing sets. Table 3.4 shows that the recall rates of SDAC in all modes are higher than 65% in the first two years, indicating that they can still detect a majority of obfuscated malware in such cases.

The category-III obfuscation methods are mostly achieved by making certain malicious codes loaded at runtime [2], which is invisible from static

Table 3.4: Robustness of SDAC (in recall rate) against Category-II Obfuscation with 2011 Training Set

Testing set	2012	2013	2014	2015	2016
SDAC-FMU	81.41%	69.87%	59.85%	49.88%	36.54%
SDAC-FEO	81.41%	67.22%	64.23%	36.49%	37.79%
SDAC-FMU-OL	77.63%	81.57%	60.98%	48.82%	47.49%
SDAC-FEO-OL	77.63%	65.49%	62.23%	58.03%	45.17%

analysis. In general, no static analysis is robust against category-III obfuscation methods. Nonetheless, SDAC can be potentially applied in dynamic analysis since the API call sequences captured in dynamic analysis can be directly used as the input to the API embedding step in SDAC. It remains interesting to test the robustness of SDAC in dynamic analysis against category-III obfuscation methods in the future.

Obfuscation by App Packing. Packing technique is also an effective method to apply obfuscation on apps and hide their codes. Various kinds of packing techniques are widely adopted by malware developers as reported in [32]. These mechanisms protect packed malware against reverse analysis and thus thwart path extracting and feature vector generating in SDAC. However, it is still potential to combine SDAC with either dynamic analysis tools or Android unpackers [32, 47, 94] against the packed malware.

3.5.2 API Semantic Extraction

Besides sequential APIs from which SDAC extracts API semantics, data dependency is another way to analyze APIs' relationship and extract their semantics. This method has been used in previous research such as DarkHazard [62].

SDAC currently focuses on malware detection based on sequential API analysis, which is complementary to data dependency analysis. Some APIs may have direct data dependency with each other but do not have any sequential

relationship. On the other hand, data dependency analysis may miss some API relationship in malware detection. For example, in an Android malware detection solution proposed by Wang et, al. in [81], it is found that data dependency information is lost in self-defined methods and thus may result in malicious behaviors being undetected. While in SDAC, these self-defined methods are collected together with their caller methods and callee methods, and then used in generating API sequences. It would be interesting to extend SDAC to cover data dependency relationships in semantic extraction in a future work.

3.5.3 Limitations

In our experiments, FlowDroid is used in the first step of SDAC for extracting API paths from Android apps. It is observed that FlowDroid fails to process 2.89% (1053/36490) benign samples and 1.74% (610/35106) malware samples among all the apps we collected originally, these samples are excluded in our experiments. Some failures are due to exceeding memory limit in the extracting (i.e., 4 GB in our experiments for API path extraction under the Soot tool). To address this problem, one may use more powerful computers with larger memory, or rely on other static analysis tools such as Amandroid [83] and Androguard [30] for API path extraction.

Another limitation in our experiments is that FlowDroid does not cover HTML5 codes, native codes, or the codes which are loaded at runtime. It is a future direction to extend SDAC to cover such codes by performing dynamic analysis.

Chapter 4

Differential Training: A Generic Framework to Reduce Label Noises for Android Malware Detection

4.1 Introduction

Machine learning-based Android malware detection has been a major research focus in recent years. Both model training and evaluation rely on a set of sample apps and their associated labels (i.e., benignware and malware). The sample apps and their labels can be either collected from malware detection websites such as VirusTotal [10] or manually examined and labeled by malware detection experts [91, 92, 19].

However, the current approach to labelling sample apps is not perfect due to a couple of reasons. First, the labels provided by malware detection websites are not always reliable [43]. To verify this, we randomly chose 50,000 APPs on VirusTotal, and downloaded their scanning reports twice in 2016.7 and 2018.7, respectively. Among them, over 10% of the samples (5310/50000) are given different labels in the two reports. On the other hand, manually labelling is often costly and time-consuming, and it is difficult to scale up to massive datasets.

The label noises in app datasets may distort Android malware detection in two main aspects.

- According to F. A. Breve. et al. [22], the noises in sample labels worsen the performance of malware detection models trained with them, making them less effective in real-world cases.

- The noisy labels used for model testing and verification misjudge the real performances of existing malware detection solutions. In the case study sections of various research papers on malware detection (e.g., [89, 62, 91]), many false positive/negative cases are reported in fact due to mislabeled samples.

The noisy label problem is intrinsic in malware detection and challenging to deal with. The situation is worse due to ever-growing sizes of datasets that are used in machine-learning based malware detection. It remains challenging to work with noisy datasets, so as to improve both training of malware detection models and their evaluations. However, this problem has not been rigorously addressed, especially in the malware detection community.

Standard benchmark datasets have been built and widely used in certain other machine-learning fields such as image processing and natural language processing, where the quality of data labels can be verified by average human users through user studies or crowdsourcing. However, it is highly challenging for domain experts in malware detection (not to mention average human users) to ensure the correctness of all data labels in a massive dataset because the techniques for composing malware are highly complicated and constantly evolving. This is one of the reasons that no universal benchmark dataset has been built for Android malware detection, making it difficult for the comparison across many malware detection approaches as they were evaluated on different datasets of unknown qualities.

Towards addressing the problem of malware detection with noisy datasets,

we propose Differential Training, a novel noisy label detection framework for machine learning based Android malware detection. We make a meaningful assumption that the whole set of apps is noisy (i.e., no individual apps' labels are known 100% correct), but a majority of sample apps are correctly labeled. Differential Training can improve the performance of any machine learning based Android malware detection approaches by reducing label noises in their datasets.

In particular, Differential Training makes use of the intermediate states of deep learning classification models during training for noisy label detection. According to Schein, etc. in [70], the intermediate states of a classification model, represented by variances of sample losses, can be used as an effective measurement on the samples' uncertainty so as to help identify those that are not predicted properly within the current model. In other research (e.g., [72, 42]), such samples are paid extra attention in training so as to accelerate the learning of models.

A fundamental assumption in the previous research mentioned above is that all samples' labels are correct. Therefore, the mismatching between desired labels and predicted labels in training is attributed to immature model training. In the case of noisy labels being present, the wrong labels also contribute to the mismatching between desired labels and predicted labels during model training.

Differential Training relies on a new heuristic, which we call *differential training heuristic*, to reduce label noises in a given set of sample apps. The heuristic differentiates between correctly labeled samples and wrongly labeled samples according to their loss values in training two deep learning classification models of the same model architecture, where one model is trained with the entire dataset, and the other is trained with a randomly down-sampled subset of the given dataset. A sample's label is considered to be "wrong" and thus revised/flipped if its loss values appear to be outliers in comparison to other samples' loss values.

This heuristic is based on an observation that correctly labeled samples tend to behave consistently in training the two classification models, while the wrongly labeled samples tend to behave differently, and thus can be detected and revised. Differential Training applies this heuristic iteratively until a convergency condition is satisfied. After this, any machine learning based malware detection approach can be trained with the set of all sample apps and their revised labels. Rigorous experiments on various datasets and malware detection approaches show that differential training is clearly effective in noisy reduction and performance improvement for machine learning based malware detection.

The main contributions of this chapter are summarized below:

- We develop a new generic framework, Differential Training, to reduce label noises for large-scale Android malware detection. Differential Training employs a novel approach to detecting noisy labels in multiple iterations according to the intermediate states of two deep learning classification models of identical architecture, one of which is trained on the whole training set of apps, and the other is trained on a randomly down-sampled set of apps. A new heuristic is proposed to distinguish between wrongly-labeled apps and correctly-labeled apps based on an outlier detection on their loss values, which are taken from the intermediate states of the two classification models.

- Differential Training enjoys high practicality because it is generic, automated, and independent to correctly-labeled datasets. Differential Training is generic as it can work with any machine learning based malware detection approach for reducing label noises of its dataset and improving its training and performance evaluation. Differential Training is fully automated in the label noise reduction process which requires neither domain knowledge nor manual inspection. In addition,

Differential Training can operate on noisily-labeled datasets only. It does not rely on any extra datasets whose labels are all correct like other noise-tolerance classification approaches such as MentorNet [42] and distilled-based learning model [48].

- The effectiveness of Differential Training is evaluated with three different Android malware detection approaches, including SDAC [88], Drebin [19], and DeepRefiner [90], as well as three different datasets, whose sizes are 69k, 129k, and 110k, respectively. Applying to these datasets, Differential Training can reduce the size of wrongly-labeled samples to 12.6%, 17.4%, and 35.3% of its original size, respectively. Consequently, Differential Training improves the performance of each malware detection approach considerably after noisy reduction is conducted to their training datasets where the noise level is about 10%. In terms of F-score measured with ground-truth data, SDAC is improved from 89.04% to 97.19% (upper bound 97.71%), Drebin from 73.20% to 84.40% (upper bound 93.34%), and DeepRefiner from 91.37% to 93.41% (upper bound 93.59%). The improved performance is relatively close to their upper bound 97.71% for SDAC, 93.34% for Drebin, and 93.59% for DeepRefiner which are trained with all correctly-labeled datasets. A similar trend is also observed at various noise levels.

- Differential Training also outperforms the state-of-the-art noise-tolerant classification solution, Co-Teaching, which is designed for robust training of deep neural networks with noisy labels [39]. Differential Training detects significantly more wrongly-labeled samples than Co-Teaching. In particular, the improvements are 10.96% (from 76.49% to 87.45%) on SDAC dataset, 4.37% (from 78.14% to 82.51%) on Drebin dataset, and 42.99% (from 21.72% to 64.71%) on DeepRefiner dataset.

4.2 Preliminaries

4.2.1 Machine Learning Based Android Malware Detection

We aim to reduce label noises for machine learning based Android malware detection that relies on a binary classification model to predict the label, which is either benign or malicious, for each given Android app. A machine learning based Android malware detection model is trained by a set of labeled Android apps (i.e., training set) in two main steps, where the first step transforms each Android app into a numerical feature vector, and the second step trains the model classifier using the apps' numeric feature vectors and their corresponding labels. After training, the model's performance can be evaluated using a set of labeled apps (i.e., testing set), based on the differences between their predicted labels and given labels.

4.2.2 Training Noise Detection Models

In the process of noisy label detection, Differential Training keeps training two identical deep learning classification models, which we call noise detection models. Each of these noise detection model is trained to classify any given sample app to be either malware or benignware. The training of each noise detection model consists of multiple epochs. In each epoch, each sample and its associated label are taken from a training dataset and fed into the model through two successive phases: forward propagation and backward propagation.

In the forward propagation phase, the feature vector of a given sample is taken as input to the noise detection model. A loss function is used to calculate a loss value for the sample according to the input vector and the parameters the noise detection model. Then, a predicted label is generated for the sample and compared to the given label of the sample.

In the back propagation phase, the gradient of the loss function with respect

to each parameter in the noise detection model is calculated. Each parameter is then updated according to the gradient and the loss value in an optimal manner so as to minimize the average loss value in the next epoch Since the model parameters are adjusted in the whole training process, the average loss value for the samples in the whole training dataset is optimized to decrease from the first epoch to the last. The number of epochs is determined by a convergency condition under which the average loss value in the last epoch is considered to be good enough.

4.2.3 Underlying Assumption

The underlying assumption made by Differential Training is that the majority of sample apps in the dataset are correctly labeled; however, it is unknown whether the label of any specific sample is correct or not. Note that it is meaningful to assume that more than 50% of the sample apps are correctly labeled since if it is not the case (i.e., the quality of dataset is even lower than random labelling), a flipping of each and every label would make this assumption valid.

Differential Training does not rely on any set of individual apps whose labels are 100% correct, which is different from other noise-tolerance classification approaches such as MentorNet [42] and distilled-based learning model [48]. It requires no manually checking on any sample apps in Differential Training, which is fully automatic in reducing label noises for Android malware detection.

4.3 Differential Training Heuristic

Differential Training trains two deep learning classification models of identical architecture iteratively for noisy label detection. We call these two models *noise detection models*, which can be any deep learning classification models to classify apps to be either benign or malicious according to the apps' feature vectors. The whole process of Differential Training consists of multiple

iterations, in each iteration two different models are trained where the first model is trained with the whole set of available training apps (and their labels), while the other is trained with a randomly down-sampled subset of the whole set. For convenience, we refer to the whole set of apps as *WS*, and the down-sampled subset as *DS*. We also refer to the first noise detection model as *WS model*, and the other as *DS model*.

In each iteration, Differential Training relies on a new heuristic, named *differential training heuristic* to reduce label noises in DS. The heuristic states that the training behaviors of correctly labeled samples across the two models are statistically different from those of the wrongly labeled samples across the two models, where the training behavior of a sample across the two models is described by the concatenation of the loss values produced for the sample in all epochs of the two models. Given the assumption that a majority of sample apps are correctly labeled, the wrongly labeled apps in DS can be detected statistically using an outlier detection on the training behaviors of all apps in DS.

Experimental Observation: The heuristic is enlightened by our observation in the following experiments. When we observe a single model, either WS model or DS model, the differences in training behaviors between correctly-labeled samples and wrongly-labeled samples are not too significant; however, when we combine training behaviors across the two models, the differences between the two types of samples become more apparent.

In the experiments of showing our observation, Differential Training is applied to a set of sample apps that are randomly collected from a public Android app sharing project [16]. To guarantee the correctness of the samples' labels, we further check their scanning results from VirusTotal, and remove all (which is equivalent to around 10%) of the samples whose scanning results ever changed since August 2016. We use 50,000 samples to build the WS set, and choose 10% of them randomly as "noises" whose labels are manually flipped.

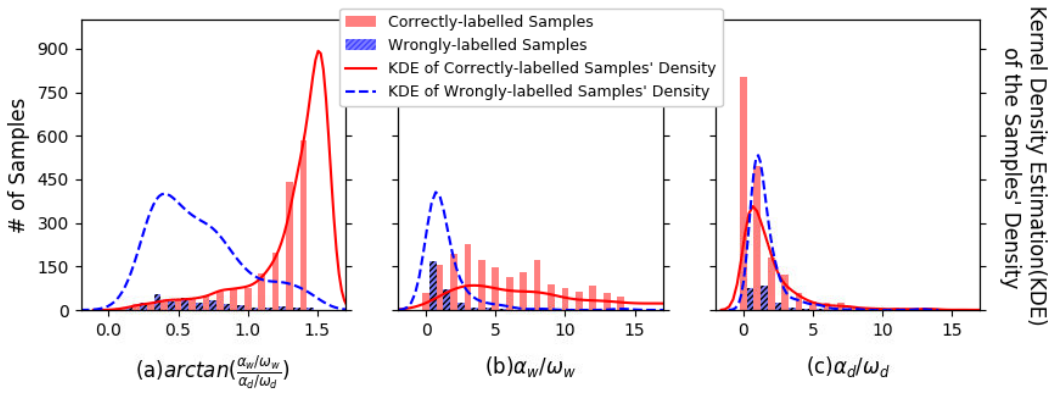


Figure 4.1: Distributions of Correctly-Labeled Samples and Wrongly-Labeled Samples

The architecture of the WS model and the DS model is chosen to be Multi-Layer Perceptron consisting of an input layer, two hidden layers, and a softmax layer, where the first hidden layer consists of 500 nodes, and the second layer consists of 1000 nodes. The training of the WS model and the DS model implements the learning rate decay and the early stopping API, and sets all hyper parameters to their default as provided in TensorFlow [8].

First, we choose 3,000 samples randomly from WS to form DS. Then we use WS to train the WS model, and use DS to train the DS model. For each sample in DS, we use α_w and ω_w to record its loss value in the first epoch and the last epoch, respectively, during the training of the WS model; and further use α_d and ω_d to denote its loss value in the first epoch and the last epoch, respectively, during the training of the DS model.

Figure 4.1 illustrates the distributions and their fitting curves of correctly-labeled samples and wrongly-labeled samples w.r.t. three variables, including $(\alpha_w/\omega_w)/(\alpha_d/\omega_d)$ in Figure 4.1 (a), (α_w/ω_w) in Figure 4.1 (b), and (α_d/ω_d) in Figure 4.1 (c). The bars in the figure indicate the number of samples at certain value of the corresponding variable, while the curves illustrate the kernel density estimation of the corresponding variable, which is a non-parametric estimation of the variable's probability density function.

Figure 4.1 shows that the differences between the correctly-labeled

Table 4.1: Wasserstein Distance between Distributions of Correctly-Labeled and Wrongly-Labeled Samples

Model(s) Used	Wasserstein Distance
WS model	0.00295
DS model	0.01047
Both WS model and DS model	0.04387

samples and the wrongly-labeled samples are more apparent in terms of their distributions measured from the loss values in training both the WS model and the DS model as shown in Figure 4.1 (a), than in training the WS model alone as shown in Figure 4.1 (b), and in training the DS model alone as shown in Figure 4.1 (c).

The distribution differences shown in Figure 4.1 between the correctly-labeled samples and the wrongly-labeled samples can be measured in Wasserstein distance [9], which quantifies the minimum "cost" of turning one distribution to another. A larger Wasserstein distance represents more significant difference between two distributions. Table 4.1 measures the Wasserstein distances between the distributions that are given in Figure 4.1. It suggests to distinguish between correctly-labeled samples and wrongly-labeled samples according to the loss values collected in training both the WS model and the DS model.

The size of DS is an important factor in differentiating the distributions between correctly-labeled samples and wrongly-labeled samples. Figure 4.2 and Table 4.2 show that using smaller DS yields greater difference; it is thus desirable to choose DS as small as possible in Differential Training.

On the other hand, DS should be large enough to converge the training of the DS model [29]. Therefore, we have two criteria for choosing the size of DS: (1) DS should be as small as possible to distinguish between correctly-labeled samples and wrongly-labeled samples, and (2) DS should be large enough for converging the training of the DS model. In our experiments, we use a grid

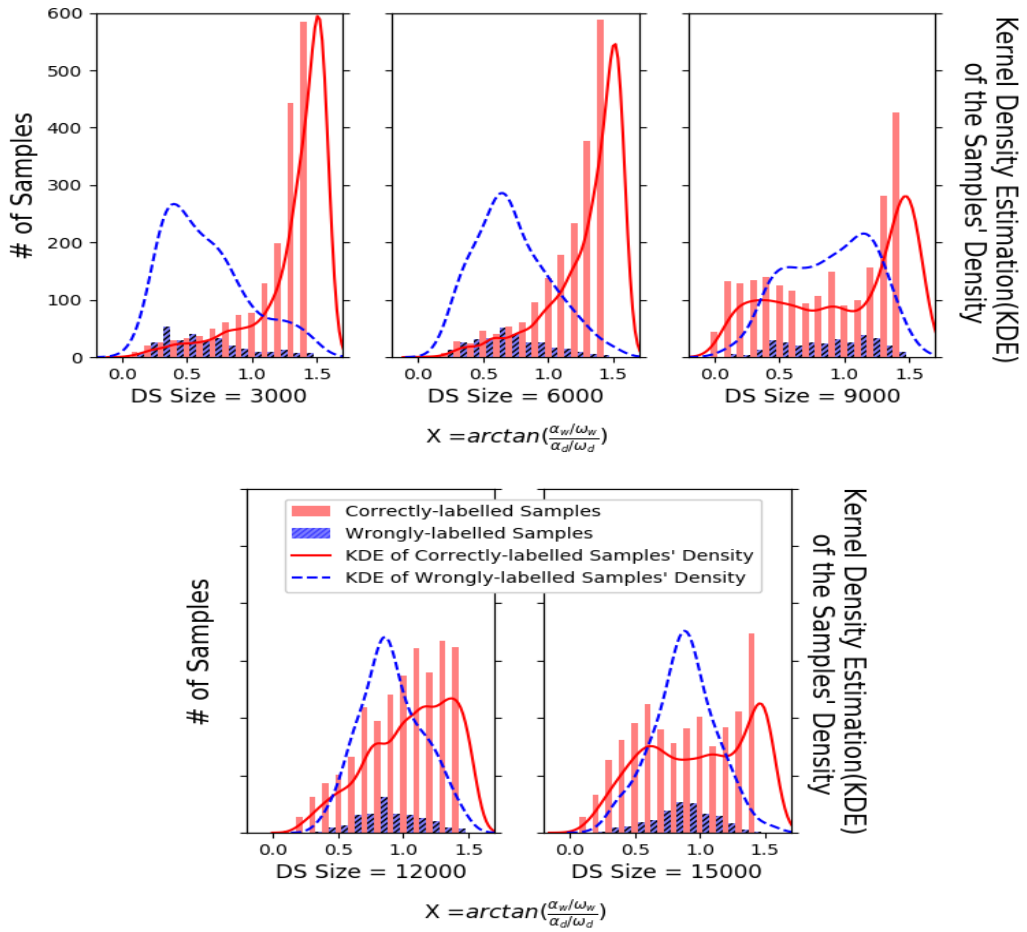


Figure 4.2: Distributions of Correctly-Labeled Samples and Wrongly-Labeled Samples with Different Sizes of DS

Table 4.2: Wasserstein Distance between Distributions of Correctly-Labeled and Wrongly-Labeled Samples with Different DS

Size of DS	Wasserstein Distance
3000	0.04387
6000	0.03401
9000	0.03334
12000	0.01965
15000	0.02703

search to choose the size of DS based on these two criteria.

Explanation on the differences of loss values: The differential training heuristic is also enlightened by the following theorem proved by S. Arora et al. in a recent research [18] on the differences of loss values between correctly-labeled samples and randomly-labeled samples during model training:

In a two-layer MLP model using ReLU activation and trained by gradient descent, when there are infinite nodes in hidden layers and the model is fully trained, the following equation holds:

$$\|\mathbf{y}-\mathbf{u}\left(k\right)\|_2=\sqrt{\sum_{i=1}^{n}\left(1-\eta\lambda_i\right)^{2k}\left(\mathbf{v}_i^\intercal\mathbf{y}\right)^2}\pm\varepsilon$$

where $\mathbf{y}=\left(y_1,y_2\ldots y_n\right)$ denotes all the labels of the n samples, $\mathbf{u}\left(k\right)$ denotes all the n predictions in the k_{th} epoch, and thus $\|\mathbf{y}-\mathbf{u}\left(k\right)\|_2$ refers to the $L2$-norm distance between the predicted labels and the true labels. Moreover, η refers to the learning rate. $\mathbf{v}_i$ refers to the orthonormal eigenvector of sample i and λ_i refers to its corresponding eigenvalue decomposed from the gram matrix H of the model, while the gram matrix H is decided by the two-layer ReLU model in the k_{th} epoch as defined in [87, 76, 31]. ε is a very small value that can be ignored.

The equation shows that under the ideal condition, $\left(\|\mathbf{y}-\mathbf{u}\left(k\right)\|_2\right)^2$ for a single sample i in epochs 1 to k is a geometric sequence which starts at $\left(\mathbf{v}_i^\intercal\mathbf{y}\right)^2$ and decreases at ratio $\left(1-\eta\lambda_i\right)^2$.

Furthermore, in section 4 of paper [18], it is proven that samples with true labels have better alignment with larger eigenvalues than samples with random labels (or wrong labels). In each epoch of model training, the square of $L2$ norm distance $\left(\|\mathbf{y}-\mathbf{u}\left(k\right)\|_2\right)^2$ thus demonstrates larger decreasing ratios $\left(1-\eta\lambda_i\right)^2$ for correctly-labeled samples than for wrongly-labeled samples.

Since the square of $L2$-norm distance $\left(\|\mathbf{y}-\mathbf{u}\left(k\right)\|_2\right)^2$ is exactly the same as the $L2$ loss function between the actual labels $\mathbf{y}$ and the predicted labels $\mathbf{u}\left(k\right)$,

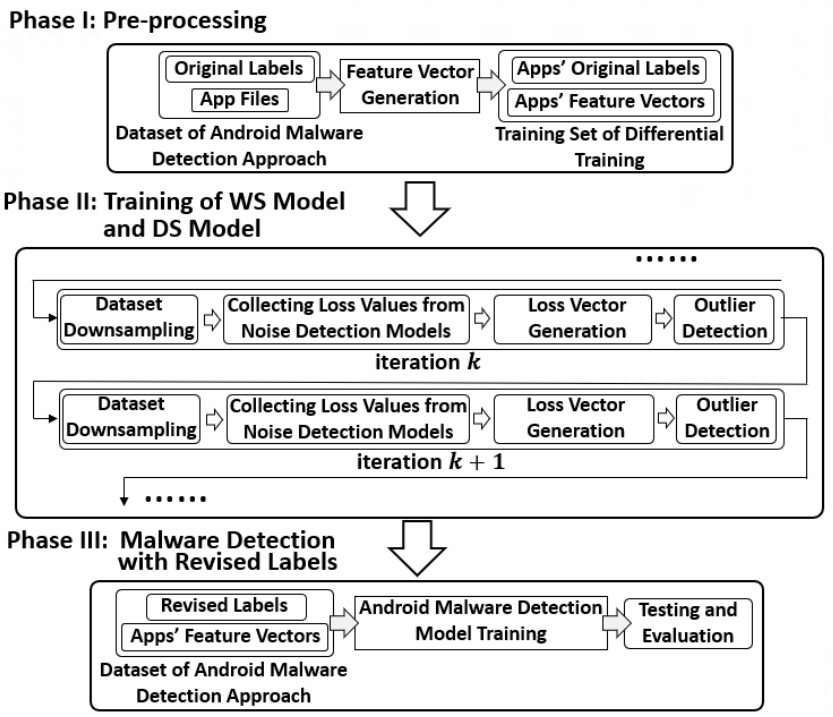


Figure 4.3: Structure of Differential Training

this theorem implies that during the training of DS/WS model, the decreasing rates of the loss values of correctly-labeled samples are larger than (and thus different from) those of wrongly-labeled samples in each epoch.

4.4 Differential Training Framework

Differential Training processes a noisy dataset in three phases: "pre-processing," "noisy label detection," and "malware detection with revised labels." The structure of Differential Training is shown in Figure 4.3.

4.4.1 Phase I: Pre-processing

In the first phase, a machine learning based malware detection approach is selected, and the raw app files from the dataset of the approach are transformed into numeric feature vectors through a "Feature Vector Generation" module which should be specified by the malware detection approach. The output of the

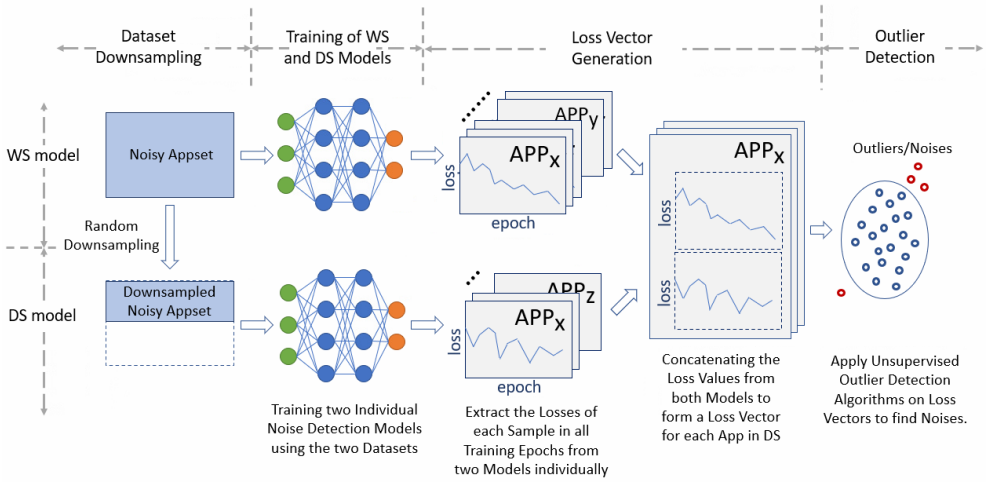


Figure 4.4: Structure of a Single Iteration in Noisy Label Detection

phase I is the whole training set WS which consists of the transformed feature vectors and their associated labels.

4.4.2 Phase II: Noisy Label Detection

The second phase "Noisy Label Detection" of Differential Training consists of multiple iterations, in each of which the noises in the training set are reduced until a stopping criterion is met. Each iteration is illustrated in Figure 4.4, which consists of four steps, including "Dataset Downsampling", "Training of WS and DS Models", "Loss Vector Generation" and "Outlier Detection".

Dataset Downsampling

In the first step "dataset downsampling", Differential Training randomly downsamples the whole training set WS to a smaller dataset, named "downsampled set," or DS for short. The size of DS is selected according to the two criteria described in the differential training heuristic.

Training of WS and DS Models

After DS is generated, two noise detection models, "WS model" and "DS model," are trained on WS and DS datasets, respectively. The two noise

detection models can be any deep learning classification models having the same network architecture for classifying apps to be either malicious or benign according to their feature vectors. Depending on the selection of app features, various deep learning classification models (e.g., Multi-Layer Perceptron, Recurrent Neural Network, and Convolutional Neural Network) may be selected to be noise detection models. Differential Training uses the noise detection models to extract the loss values for each input app during training; any deep learning classification model can be used as a noise detection model as long as it outputs a loss value for each input app in each epoch during its training process.

In our experiments conducted in this chapter, we choose the noise detection models to be Multi-Layer Perceptron (MLP) that consists of two hidden layers, where the first hidden layer consists of 500 nodes and the second hidden layer 1000 nodes, and followed by a softmax layer as output. We use the TensorFlow toolkit [8] to train the two models, where all the parameters are set to their default values in the toolkit.

Loss Vector Generation

In the third step, the loss values of each app in DS are collected from the two noise detection models during their trainings. The loss values collected from each model are arranged into a sequence in the order of training epochs. Then the two sequences are concatenated to form a "loss vector" for each app in DS.

Outlier Detection

In this step, a set of unsupervised outlier detection algorithms are applied to the loss vectors of all the apps in DS. For each app whose loss vector is detected as an outlier, its label is considered to be "wrong", and thus flipped with a probability. Several points on the outlier detection are clarified below:

- Most outlier detection algorithms require a "containment rate" parameter

Table 4.3: List of Outlier Detection Algorithms used in Differential Training

Angle-based Outlier Detector (ABOD)
Auto Encoder
Clustering Based Local Outlier Factor (CBLOF)
Histogram-based Outlier Detection (HBOS)
IsolationForest Outlier Detector (I-forest)
k-Nearest Neighbors Detector (kNN)
Local Outlier Factor (LOF)
Outlier Detection with Minimum Covariance Determinant (MCD)
Single-Objective Generative Adversarial Active Learning (So-gaal)
One-class SVM detector
Stochastic Outlier Selection (SOS)
Principal Component Analysis Outlier Detector (PCA)
EllipticEnvelope

as their input. This parameter works as a threshold in identifying outliers. In Differential training, this parameter is set to the current ratio of wrongly-labeled samples in WS. This noise ratio is estimated using the method proposed by Goldberger [38]. In particular, the noise ratio is estimated to be $(1 - a_{WS})$, where a_{WS} is the accuracy of the WS model in 5-fold cross-validation on WS.

- To avoid any bias of a single outlier detection algorithm, we use 13 different outlier detection algorithms and apply a majority voting to get the final result of outliers. Table 4.3 shows the outlier detection algorithms used in Differential Training, where the first 12 algorithms are taken from a public toolkit named "PyOD" [6], while the last algorithm "EllipticEnvelope" is taken from the toolkit "sklearn" [7].

- Another parameter named dropout ratio is introduced in this step. After each outlier is detected according to the majority voting, the label of the corresponding sample is revised/flipped with a probability equal to the dropout ratio. The dropout ratio is used to reduce the impact caused by any accidental error from either outlier detection or noise rate estimation. The use of this dropout ratio is inspired by the random dropout mechanism in neural networks training [73], and the ratio is set to 0.5 in our experiments.

Stopping Criterion

To stop the iterations in training the WS model and the DS model, we use a stop criterion that is similar to the early stopping adopted in neural network training. The iterations stop once the fluctuation of the estimated noise ratios in the last several iterations turns to be smaller than a certain threshold. In experiments, we enforce the stopping criterion through the API $earlystop\_callback()$ from the TensorFlow toolkit, where all parameters are set to their default values.

4.4.3 Phase III: Malware Detection with Revised Labels

Once the iterations stop, the apps and their associated labels in the whole set WS are ready for Android malware detection. The original Android malware detection approach can be trained using these apps' feature vectors (which were extracted in phase I) and their labels (which were revised in phase II). In the experiments below, we use ground-truth data to evaluate the performance of Differential Training with three different Android malware detection approaches, including SDAC [88], Drebin [19], and DeepRefiner [90].

We measure the performances of Differential Training using the following metrics: (i) The number and the percentage of wrong labels in the training set being reduced by Differential Training. (ii) The F-scores of the malware detection approach when it is applied to the noisy training set, the noise-reduced training set processed by Differential Training, and the "ground-truth" training set. The differences between these F-scores[1] show that how much improvement in the performance of the malware detection approach is made due to Differential Training, and how close is the improved performance to the upper bound.

In evaluating Differential Training with a malware detection approach, we

[1] A F-score is the harmonic mean between precision and recall, where precision measures the percentage of true malware among the detected malware, while recall measures the percentage of true malware being detected.

partition its "ground-truth" dataset into two parts: 80% of them are used as the training set, and the other 20% are used as testing/validation set. Given a noise ratio $0 \leq r_{noise} \leq 0.5$, we randomly select each app in the training set with probability $r_{noise} * 100\%$, and flip the labels of the selected apps to generate a noisily-labeled dataset. The default value of the noise ratio is set to 10%, which is similar to the ratio which we observed from VirusTotal during a period of three years. After Differential Training revises the labels in the training set, we refer it as the processed dataset. While we train a malware detection approach using either ground-truth dataset, noisy dataset, or processed dataset, we always evaluate its performance using a "ground-truth" testing dataset.

Without confusion, we also call the framework Differential Training by excluding phase III if the objective is to detect or reduce noisy labels in a dataset without testing the malware detection approach.

4.5 Differential Training with SDAC

In this chapter, Differential Training is firstly evaluated with the SDAC solution proposed in chapter 3. While the dataset used in this evaluation is also the same as that in section 3.4.

4.5.1 Performance of Differential Training with SDAC

Table 4.4 summarizes SDAC dataset and SDAC performances, which are measured on the correctly-labeled dataset, the nosily-labeled dataset, and the noise-reduced training set processed by Differential Training. Overall, Differential Training revise 5,246 labels correctly, revise 343 labels wrongly. The percentage of noise labels thus reduces from 9.91% to 1.26% in the nosily-labeled dataset due to the process of Differential Training.

More details are given in Figure 4.5 which shows the number of labels that are revised correctly by Differential Training (i.e., true positives), and the

Table 4.4: The Evaluation of Differential Training with SDAC

Android Malware Detection Approach		SDAC	
# Samples in the Whole Dataset		69,933	
# Benignware	# Malware	35,437	34,496
# Samples in Noisy Training Set		56,650	
# of Noises added		5,614 (9.91%)	
# detected TP	# detected FP	5,246	343
# of Remained Noises in Processed Dataset		711 (1.26%)	
F-score with Correctly-laballed Dataset		97.71%	
F-score with Noisily-laballed Dataset		89.04%	
F-score with Processed Dataset		97.19%	

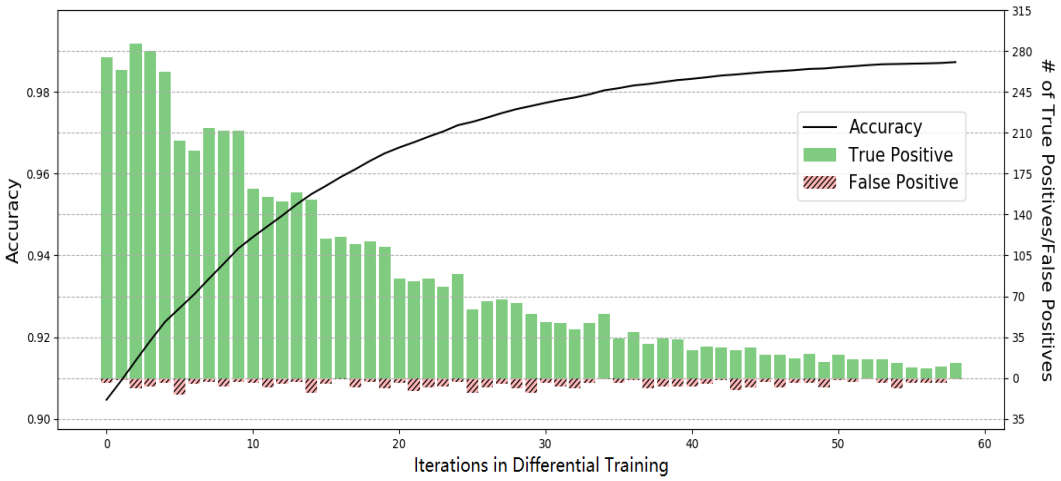


Figure 4.5: Noise Reduction on SDAC Dataset

number of labels that are revised wrongly (i.e., false positives) in each iteration. It also shows the accuracy of the revised labels (i.e., the percentage of the revised labels that are correct) in each iteration. When Differential Training converges, the accuracy of the revised labels reaches close to 99%.

The F-score of SDAC improves from 89.04% to 97.19% after noise reduction on the training set, and this improved F-score is very close to its upper bound 97.71% (see Table 4.4). These results show that Differential Training can greatly reduce the number of wrong labels in the training set, and improve the performance of Android Malware detection approach due to the use of noise-reduced training set in training.

4.5.2 Runtime Performance of Differential Training with SDAC

The total time cost of Differential Training with SDAC in this experiment is about 55.34 hours. In detail, Differential Training performs 58 iterations; each iteration takes about 57.3 mins on average, which includes 50.6 mins spent on training the WS model, and less than 4 mins spent on training the DS model. The rest of time in each iteration is used for performing outlier detection and noise ratio estimation.

4.6 Differential Training with Drebin

4.6.1 Introduction of Drebin

Drebin [19] is a lightweight Android malware detection solution published in 2014 based on static analysis. Drebin extracts the following features from each app: hardware components, required missions, App components, filtered intents from Android manifest files, critical API calls, actually used permissions, human-defined suspicious API calls, and network address strings from disassembled codes. Drebin converts each app into a feature vector of 545,433 dimensions. It relies on a linear support vector machine (SVM) classifier for malware detection, and uses its linear weights for identifying the features that make significant contributions to malware detection.

Compared to SDAC that was published recently in 2020, Drebin is more classic which has been cited frequently in malware detection research since 2014. In addition to evaluate the effectiveness of Differential Training on SDAC, we also test it on the classic Drebin with relatively old dataset.

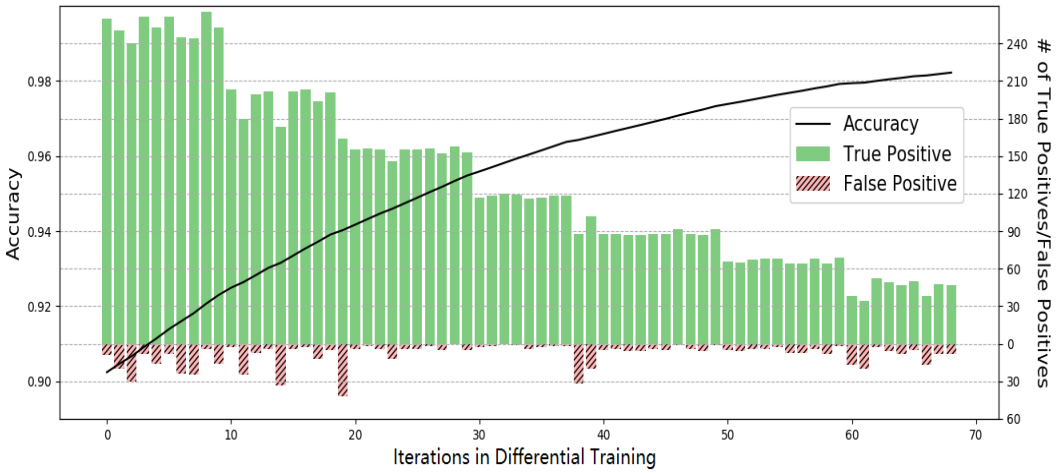


Figure 4.6: Noise Reduction on Drebin Dataset

4.6.2 Drebin Dataset

Drebin was evaluated on a dataset collected by 2014, which was composed of 5,560 "malware samples" and 123,453 "benign apps." The Drebin dataset has been frequently used in malware research since its publication. We checked the dataset in June 2019 using VirusTotal to guarantee the ground-truth of the dataset. In detail, we found no contradictory labels in the dataset except that some apps were too old to receive any report. Compared to the SDAC dataset where malware takes up 49.3% of all apps, the Drebin dataset is highly imbalanced as malware samples account for 4.3% of all apps. This is another reason that we choose Drebin so that we can test Differential Training on a highly imbalanced dataset.

4.6.3 Performance of Differential Training with Drebin

Figure 4.6 shows the performance of Differential Training in each iteration, where the red bars and green bars are measures of true positives and false positives, respectively. In total, 9,121 noisy labels are detected and revised correctly by Differential Training, and 605 labels are detected and revised mistakenly. The accuracy of Differential Training for label revisions converges close to 98%.

Table 4.5: The Evaluation of Differential Training with Drebin

Android Malware Detection Approach		Drebin	
# Samples in the Whole Dataset		129013	
# Benignware	# Malware	123,453	5,560
# Samples in Noisy Training Set		103,210	
# of Noises added		10,009 (9.70%)	
# detected TP	# detected FP	9,121	605
# of Remained Noises in Processed Dataset		1805 (1.75%)	
F-score with Correctly-laballed Dataset		93.34%	
F-score with Noisily-laballed Dataset		73.20%	
F-score with Processed Dataset		84.40%	

Table 4.5 shows that Drebin's performance improves from 73.20% to 84.40% in F-score if it is trained on the processed dataset, for which Differential Training reduces the percentage of noise labels from 9.70% to 1.75%. Compared to the original F-score, the improved F-score is closer to its upper bound 93.34% which is achieved by Drebin trained with the correctly-labeled training set.

4.6.4 Runtime Performance of Differential Training with Drebin

The total time cost of Differential Training with Drebin in this experiment is about 62.52 hours, which converges in 68 iterations. Each iteration takes 55.2 minutes on average while the training of the WS model takes 52.0 mins and the training of the DS model takes 2.8 mins.

4.7 Differential Training with DeepRefiner

4.7.1 Introduction of DeepRefiner

DeepRefiner [90] is a Android malware detection approach that connects two deep learning models in sequential. The first model is a Multi-Layer Perceptron model which can detect "most significant" malware samples efficiently based

Table 4.6: The Evaluation of Differential Training with DeepRefiner

Android Malware Detection Approach		DeepRefiner	
# Samples in the Whole Dataset		110,440	
# Benignware	# Malware	47,525	62,915
# Samples in Noisy Training Set		88352	
# of Noises added		8,835 (10.00%)	
# detected TP	# detected FP	7,497	2,230
# of Remained Noises in Processed Dataset		3,118 (3.53%)	
F-score with Correctly-laballed Dataset		93.59%	
F-score with Noisily-laballed Dataset		91.37%	
F-score with Processed Dataset		93.41%	

on XML files in app APK packages. The second model is a long short-term memory model which detects "more advanced" malware samples from those apps for which the first model cannot provide reliable classification results. The second model relies on checking the semantic structures of Android bytecodes in malware detection.

According to [90], the first model of DeepRefiner can be used alone and it achieves 87.3% accuracy in malware detection on a dataset of 110,440 apps. We choose this model to evaluate how Differential Training performs if the malware detection approach is not extremely accurate but very efficient.

4.7.2 DeepRefiner Dataset

The original DeepRefiner dataset consists of a set of benign applications that were collected from Google Play as well as a set of malicious apps that were collected from VirusShare and MassVet in 2015 to 2016. We then downloaded their scanning reports from VirusTotal in June 2019 and removed all the samples with different labels. The remaining dataset contains of 62,915 malicious applications and 47,525 benign applications that were collected in 2016.

4.7.3 Performance of Differential Training with DeepRefiner

Figure 4.7 shows that after being processed by Differential Training, the percentage of samples with correct labels in the DeepRefeiner dataset increases

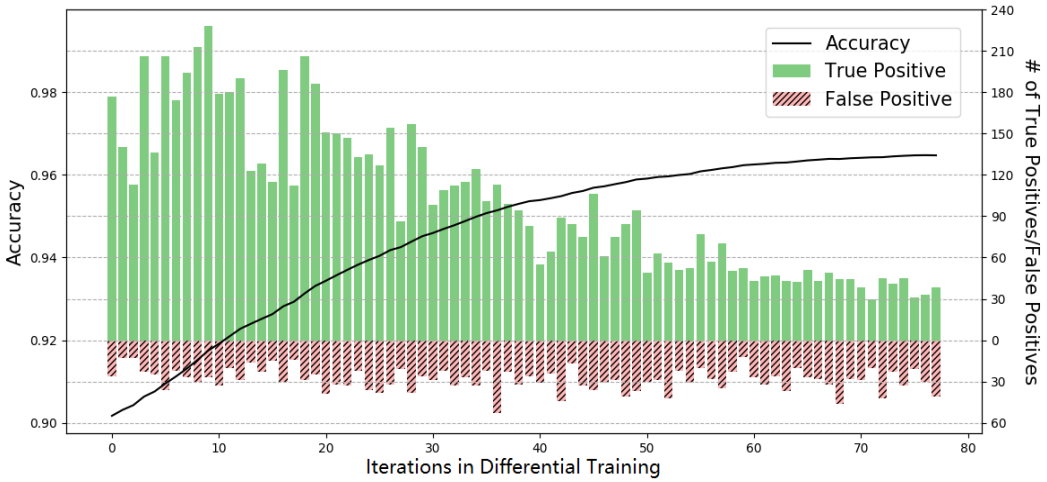


Figure 4.7: Noise Reduction on DeepRefiner Dataset

from 90% to 96%, while Table 4.6 further shows that 7,497 wrong labels are revised correctly, while 2,230 correct labels are revised mistakenly.

In total, Differential Training reduces 64.7% of the wrong labels and increases the F-score of DeepRefiner from 91.37% to 93.41%, which is only 0.18% lower than the F-score of DeepRefiner trained with the correctly-labeled dataset.

4.7.4 Runtime Performance of Differential Training with DeepRefiner

In the Differential Training with DeepRefiner, the total time cost is about 102.12 hours, which includes 77 iterations. The time cost for each iteration is about 79.6 minutes on average, including 70.0 minutes for training the WS model and 7.9 minutes for training the DS model.

4.8 The Impact of Noise Ratio to Noise Reduction

Differential Training is effective in reducing label noises on three different datasets at noise ratio 10% as shown in the previous sections. In this section, we further investigate the impact of noise ratio to noise reduction. For this purpose,

Table 4.7: Noise Reduction on SDAC Dataset at Different Noise Ratios

Noise Ratio	5%	10%	15%	20%	30%	45%
# of Training Set	56,550	56,550	56,550	56,550	56,550	56,550
# of Wrongly-labeled Samples	2,833	5,614	8,497	11,330	16,995	25,493
# of TP Noise Detection Results	2,540	5,246	7,977	10,465	15,762	21,858
# of FP Noise Detection Results	281	343	472	377	655	4,500
% of Noise Reduced	79.73%	87.34%	88.33%	89.04%	88.89%	68.09%
# of Wrongly-labeled Samples Left	574	711	992	1,242	1,888	8,135%

Table 4.8: Noise Reduction on Drebin Dataset at Different Noise Ratios

Noise Ratio	5%	10%	15%	20%	30%	45%
# of Training Set	103,210	103,210	103,210	103,210	103,210	103,210
# of Wrongly-labeled Samples	5,160	10,321	15,482	20,400	30,936	46,445
# of TP Noise Detection Results	4,658	9,121	12,700	17,870	26,120	44,804
# of FP Noise Detection Results	788	605	878	2,232	1,875	20,247
% of Noise Reduced	75.00%	82.51%	76.36%	76.66%	78.37%	52.87%
# of Wrongly-labeled Samples Left	1,290	1,805	3,660	4,762	6,691	21,888

we produce datasets at 5%, 10%, 15%, 20%, 30%, and 45% noise ratios, and apply Differential Training on such datasets with different Android malware detection approaches.

Note that 45% noise ratio is close to the upper bound of noise ratio (i.e., 50%)[2], indicating a data quality that is close to random labelling.

Table 4.7 shows the noise reduction results of Differential Training on SDAC dataset at various noise ratios. In these experiments, the percentage of wrong labels being reduced by Differential Training ranges from 79.73% to 89.04% as the noise ratio changes from 5% to 30%, indicating that the effectiveness of Differential Training is stable in these experiments. When the noise ratio

[2]If the noise ratio is greater than 50%, a flipping of each and every label would turn the noise ratio below 50%.

Table 4.9: Noise Reduction on DeepRefiner Dataset at Different Noise Ratios

Noise Ratio	5%	10%	15%	20%	30%	45%
# of Training Set	88,352	88,352	88,352	88,352	88,352	88,352
# of Wrongly-labeled Samples	4,415	8,835	13,253	17,670	26,502	39,758
# of TP Noise Detection Results	3,985	7,947	12,406	14,737	21,707	23,604
# of FP Noise Detection Results	1,247	2,230	3,677	5,150	8,799	14,795
% of Noise Reduced	62.02%	64.71%	65.86%	54.26%	48.71%	22.15%
# of Wrongly-labeled Samples Left	1,677	3,118	4,524	8,083	13,594	30,949

in dataset is set to 45%, which is close to the upper bound (i.e., noise ratio for random labelling), Differential Training reduces 68.09% of wrong labels. Though this percentage is lower than the other cases in the experiments, it is still substantial in this extreme case.

Table 4.8 shows a similar trend for Differential Training's effectiveness on noise reduction working on Drebin dataset at various noise ratios. The percentage of noisy labels being reduced fluctuates from 75.00% to 82.51% if the noise ratio varies between 5% and 30%, and it decreases to 52.87% at noise ratio 45%.

Table 4.9 shows a wider fluctuation margin of noise label detection rate on DeepRefiner dataset, which varies from 48.71% to 65.86% for the noise ratio range of 5% to 30%. This wider fluctuation margin is probably due to the relatively simple feature set selected by DeepRefiner as compared to more comprehensive feature sets used by SDAC and Drebin. Nonetheless, Differential Training can still detect nearly half of wrong labels even if the noise ratio is as high as 30% in the training set. While in the extremely noisy case for 45% of noise ratio, Differential Training reduces 22% of wrong labels. While this result is lower than the other cases as the noise ratio is close to random labelling, the effectiveness of Differential Training is still non-negligible in reducing the noise in the training dataset.

Tables 4.7, 4.8, and 4.9 also show a trend between the detection performance of Differential Training and the noise ratio in the dataset. When the noise ratio ranges from 5% to 30%, the detection accuracy does not change much; while the ratio increases to 45%, a significant decrease in detection accuracy is observed.

Differential Training detects label noises as outliers. When there are almost the same number of outliers as non-outliers, it is difficult for Differential Training to distinguish between outliers and non-outliers, leading to a significant drop in its detection accuracy.

4.9 Comparison among Differential Training, Co-teaching, and Decoupling on Noise Reduction

In this section, we compare Differential Training with two state-of-the-art unsupervised algorithms, including Co-Teaching and Decoupling. Both of which are designed for robust training of deep neural networks with noisy labels.

Co-Teaching trains two neural networks simultaneously on a noisy dataset, where the two models are of identical architecture but initialized independently at the beginning. Given each mini-batch of the dataset, each network views its small-loss data samples as potentially-clean samples, and provides them to its peer network for updating the parameters in the peer network. In Co-Teaching, the two networks have different learning abilities; they can thus filter each other's different types of error that are introduced by noisy labels in the learning process. After training, the two fully-trained models cooperate to output a predicted label for each sample in the testing phase, where the predicted label is determined by an output weight that is the sum of the output weights of the input sample from the two models. Co-Teaching can be used to detect noisy labels. If the predicated label of an input sample is different from the original label of the input sample, the original label is considered as a noisy label, and thus flipped. The source code of this algorithm is publicly available and provided in [4].

Decoupling also trains two neural networks simultaneously on a noisy dataset, with these two networks being of the same structure but initialized independently. During the model training, each mini-batch of the dataset is fed to both models simultaneously to generate the prediction results. If a sample in the mini-batch is predicted with different labels from the two models, it is regarded as meaningful for the model learning and only the "meaningful" samples in the mini-batch are later used in the backward propagation step to update the parameters in both models. At the end of training, Decoupling randomly chooses one of the two trained models as the produced classifier.

Decoupling can be used to detect noisy labels. If an input sample's label that is predicted by the produced classifier is different from its original label, the original label is considered as noisy, and thus flipped for correction. Decoupling's source code is publicly available and provided in [5].

We apply both Differential Training, Co-Teaching, and Decoupling to the noisy versions of all three datasets, including SDAC dataset, Drebin dataset, and DeepRefiner dataset, where the noise ratio is set to 10% as in the default setting. We compare their performances in terms of the percentage of wrongly labels being detected/reduced in the noisy datasets. For fair comparison, the neural networks used in Co-Teaching or Decoupling are chosen to be the same as the ones used in Differential Training, being two-layer MLP networks with 500 nodes in the first layer and 1000 nodes in the second layer.

Table X compares Differential Training with Co-Teaching and Decoupling in terms of noise detection result and runtime performance. The runtime performance is evaluated on a single desktop personal computer without GPU. The PC is equipped with one Intel(R) i5-4590 3.3 GHz CPU and 12 GB physical memory running on the Ubuntu 14.04 (LTS) operating system.

The table shows that while Differential Training takes longer time than Co-teaching and Decoupling, it outperforms these two approaches considerably in all three cases in terms of noise detection accuracy. Specifically, Differential Training produces the most True-Positive (TP) results and the least False-Positive (FP) results in all the cases except for the FP result in the case of DeepRefiner. The malware detection results (F-score) with the datasets processed by Differential Training are also better than those processed by the other two noise detection approaches.

Different strategies are exploited for identifying or processing noise samples. Differential Training identifies a sample as "noise" based on all of its loss values in the whole training process, while Co-Teaching treats a sample to be potentially-clean based on its individual loss value in each mini-batch, and

Table 4.10: Comparison between Differential Training (DT), Co-Teaching (CT) and Decoupling (DC)

	SDAC Dataset	Drebin Dataset	DeepRefiner Dataset
% of Wrongly Labels Reduced by DT	87.45%	82.51%	64.71%
% of Wrongly Labels Reduced by CT	76.49%	78.14%	21.72%
% of Wrongly Labels Reduced by DC	68.43%	65.37%	29.80%
# of TP/FP Noises Detected by DT	5,246/343	9,121/605	7,497/2,230
# of TP/FP Noises Detected by CT	5,131/841	8,997/933	3,311/1,392
# of TP/FP Noises Detected by DC	4,305/463	8,107/1,360	4,392/1,606
Detection results (F-score) on Noisy dataset	89.04%	73.20%	91.37%
Detection results (F-score) on dataset processed by DT	97.19%	84.40%	93.41%
Detection results (F-score) on dataset processed by CT	96.01%	77.36%	93.33%
Detection results (F-score) on dataset processed by DC	92.38%	79.80%	92.19%
Runtime Performance of DT (hour)	55.34	62.52	102.12
Runtime Performance of CT (hour)	4.08	20.09	23.93
Runtime Performance of DC (hour)	2.24	21.71	14.15

Decoupling identifies a noise sample based on its prediction result that is related to its loss value in the last epoch only. Our comparison shows that the strategy exploited by Differential Training is more reliable in detecting noisy labels than other strategies exploited by Co-Teaching and Decoupling.

4.10 Discussion

4.10.1 Limitation

Time cost of Differential Training: As shown in section IX, Differential Training has a higher time cost than Co-teaching and Decoupling. This is mainly because Differential Training reduces the label noises in a gradual way in multiple iterations, while in each iteration two separate classification models (i.e., WS model and DS model) are trained. In comparison, both Co-teaching

and Decoupling rely on two classification models being trained in a single iteration.

Accuracy of noise ratio estimation: The accuracy of the noise ratio estimation used in outlier detection may affect the performance of Differential Training. If the estimated noise ratio is significantly different from the actual ratio, Differential Training may produce more false positives and false negatives in identifying noise labels. While the noise ratio estimation algorithm we adopted enables Differential Training to outperform Co-Teaching and Decoupling in rigorous experiments, it is still possible to further improve the performance of Differential Training by adopting a more accurate noise ratio estimation algorithm in its outlier detection. We leave this as a future work.

4.10.2 Generalization on Differential Training

Differential Training is designed to identify and correct wrongly-labeled data samples for Android malware detection. In this chapter, we consider Android malware detection as a binary classification problem, where each app is labeled either benign or malicious. While we expect that the idea of Differential Training can be generalized to other fields for *identifying* data samples whose labels are misclassified, it cannot be directly applied to *correcting* wrong labels for multiclass classification. This is because Differential Training simply flips the labels for identified noise samples to correct them. For multiclass classification, additional effort is to be made on how to correct wrong labels.

Chapter 5

Dynamic Attention: A Noise-Tolerant Dynamic Analysis Approach to Android Malware Detection based on Attention Variances

5.1 Introduction

Machine learning-based Android malware detection has been a major research focus in recent years. In this area, dynamic analysis approach is always a hot topic and attracts attention in both academic and industry. Many reseachers have published their dynamic analysis approaches to Android malware detection (e.g., [68, 55, 45]), while companies like Google and Huawei also deploy dynamic analysis systems for Android malware detection on their app markets or mobile devices [12]. Compared with static analysis approaches, dynamic analysis approaches are more robust against anti-detection methods such as repackaging, obfuscation, and dynamic loading [58].

We focus on supervised learning where accurate and reliable ground-truth data samples (and their labels) are crucial. However, the labels of the feature vectors of malicious apps in dynamic analysis may be mistaken due to the imperfect trigger procedures in the feature collection step. In particular, the feature triggering modules and scripts such as [81, 84] are designed to emulate the real users' inputs or events, and record the behaviors of apps as their feature vectors for malware detection in dynamic analysis. However, no triggering modules and scripts can perfectly trigger all potential malicious behaviors. For example, the logic bomb [78, 65] is a technique causing apps to suspend specific behaviors if they are in dynamic analysis environments.

Therefore, the triggered behavior traces collected from samples labelled as "malware" may not contain "malicious" behaviors. Thus the feature vectors extracted from the malware samples/apps may be mislabelled in the model training process. Using such wrong labels in training downgrades the performance of these trained malware detection models.

The noisy label problem is intrinsic in the dynamic analysis approaches to Android malware detection and the situation is worse due to ever-growing sizes of datasets that are used in machine-learning based malware detection approaches. It is nearly impossible even for domain experts to manually analyze the correctness of labels in the training dataset because the behavior traces produced in dynamic analysis are usually of great lengths of more than 100,000 and the techniques for composing malware are also highly complicated and constantly evolving.

Towards addressing the noise problem caused by imperfect trigger procedures, we propose Dynamic Attention, a novel noise-tolerant dynamic analysis approach to Android malware detection. Dynamic Attention leverages the attention mechanism to detect noisy labels associated with the feature vectors of malicious apps in its training phase. The attention mechanism [77] works as an inductive module in machine learning that discovers the relations

between input values, and allows a prediction model to look over all the values in the input vector and figure out the most important parts in the input vector that lead to the prediction output.

We make a meaningful *asymmetric assumption* that the labels "benign" of the behavior traces (or feature vectors) that are triggered from benign apps are correct since these apps perform no malicious behaviors, while the labels "malicious" of the behavior traces (or feature vectors) triggered from malicious apps are noisy. It is unknown which "malicious" labels are correct or wrong for the behavior traces of malicious apps.

Moreover, we further introduce a new heuristic, named *dynamic attention heuristic*, for noise label detection. The dynamic attention heuristic relies on the variances of the attention weights of the feature vectors that are derived from malicious apps to detect noisy labels during the training of deep learning based malware detection models. If a feature vector labelled as "malicious" has high variance among its self-attention weights, the dynamic attention heuristic regards its label as more trustworthy (i.e., it is likely that malicious behaviors are triggered out); otherwise, its label is less trustworthy (i.e., it is likely that no malicious behavior is trigged out).

In deriving this heuristic, we observe that malicious behaviors, if triggered out, usually represent a small portion of the entire behavior traces. This is because malware developers tend to shorten the malicious code so that it is easier to hide malicious operations among benign ones. On the other hand, the triggering modules and scripts in dynamic analysis are designed to cover app codes as much as possible such that the extracted feature vectors are lengthy.

We deploy the attention mechanism on each input feature vector to calculate the variance of self-attention weights. If a feature vector's label is "malicious" and malicious behaviors have been indeed triggered out, the small portion representing malicious behaviors is given high self-attention weights by the attention mechanism, while the rest majority portion is given relatively low

weights. Consequently, the variance among the attention weights in the input vector is relatively high.

In comparison, if a feature vector is labelled as "malicious" but does not contain any malicious behavior due to imperfect triggering. The attention mechanism cannot locate which part of the feature vector represents malicious behaviors, and thus assign similar weights to all the values in the input vector. The variance among the attention weights in the input vector is relatively low.

After the calculation of the attention variances for all input feature vectors in the training phase, the training weights of those feature vectors having higher attention variances are increased, while those with lower attention variances are decreased for the training of the malware detection model. The trained model is more tolerant to noise labels because it learns more from the feature vectors which labels are more trustworthy and learns less from the feature vectors which labels are less trustworthy.

The main contributions of this chapter are summarized below:

- We develop Dynamic Attention, a noise-tolerant dynamic analysis approach to Android malware detection. Dynamic Attention deploys an additional step during the model training to reduce the impact caused by label noises. This step leverages the attention mechanism to calculate the self-attention weights of feature vectors and the variances of the attention weights. In the training of an LSTM model for Android malware detection in dynamic analysis, Dynamic Attention assigns high (low, respectively) weights to the feature vectors that are extracted from malicious apps if their attention variances are high (low, respectively). To the best of our knowledge, Dynamic Attention is the first noise-tolerant dynamic analysis approach to Android malware detection.

- Dynamic Attention enjoys high practicality due to full automation and its independence to correctly-labeled datasets. Dynamic Attention is

fully automated since it relies on neither domain knowledge nor manual inspection in its training. Besides, Dynamic Attention works on any noisily-labelled dataset in dynamic analysis; unlike MentorNet [42] and distilled-based learning model [48], Dynamic Attention does not need to rely on any extra dataset whose labels are all correct. It thus spares the costly work for examining malicious behaviors from the behavior traces that are extracted from malicious apps.

- The effectiveness of Dynamic Attention is evaluated with a dataset containing 21,987 benignware and 33,588 malware samples. The dataset and the behavior traces extracted from the apps in the dataset were provided by an industry company in collaborating with a research institute beyond the control of our research. Compared to a vanilla neural network model, Dynamic Attention improves the F-score of detection accuracy from 63.05% to 82.13%.

5.2 Preliminaries

5.2.1 Machine Learning Based Android Malware Detection with Dynamic Analysis

We aim to build a noise-tolerant Android malware detection approach which relies on a binary classification model to predict the label of any candidate sample, which can be either benign or malicious.

In general, a dynamic-based Android malware detection model is trained by a set of labelled apps(Training set) in four steps: First, the label of each app in the set is collected in advance (In this chapter, we name the app's label as "static label"). Then each app will be installed on emulators or devices, and a module is employed to generate a series of inputs to the emulators or devices to simulate the human activities to the app. Meanwhile, another module is deployed at the

same time to record the behaviors in the forms of data flow, API calls, system calls, etc. In the third step, the collected behaviors are transformed into feature vectors according to the feature set designed in the detection approach. Finally, the "static label" for the app is assigned to the feature vector's label, then feature vectors of all the app samples together with corresponding vector labels are fed to the classification model to train classifiers. For simplicity, we denote the labels assigned to the dynamic-based feature vectors as "dynamic label".

After the model training, a set of labelled apps (i.e., testing set) are used to evaluate the performance of the generated model based on the differences between the predicted and actual labels of these apps.

5.2.2 Training of Neural Network Models

The training of the neural network classification models for our approach is composed of multiple epochs. In each epoch, each sample and its associated label(dynamic) are paired from the training dataset and a batch of such pairs are then fed into the model together through two successive phases: forward propagation and backward propagation.

In the forward propagation phase, the feature vector of a given sample is taken as input to the noise detection model. A loss function is used to calculate a loss value for the sample according to the input vector and the parameters of the noise detection model. Then, a predicted label is generated for the sample and compared to the given label of the sample.

In the backward propagation phase, the gradient of the loss function with respect to each parameter in the noise detection model is calculated. Each parameter is then updated according to the gradient and the loss values optimally to minimize the average loss value in the next epoch.

With the model parameters being constantly updated during the whole training process, the average loss value for the samples in the whole training

dataset is optimized to decrease from the first epoch to the last. The number of epochs is determined by a convergency condition under which the average loss value in the last epoch is considered to be good enough.

5.2.3 Attention Mechanism

Attention works in form of a component in a neural network's architecture, which can be used to quantify the interdependence among the values of the input vector and the output vector. Given an input vector, the attention component can map the most important and relevant values in the input vector for each value in the output and assign higher weights to the important values to enhancing the prediction accuracy.

In Android malware detection research, the predicted results from the classification models are often in form of one-hot vectors with the length of two, representing that the input's dynamic label being either "benign"(The output vector here is [0, 1]) or "malicious"(The output vector here is [1, 0]). The attention weights for each value in the input vector will thus demonstrate how much the selected values contribute to the label of apps.

5.2.4 Underlying Assumption

The underlying assumption made in our approach is that, in the dynamic-based analysis, the dynamic label of feature vectors triggered from app samples with "benign" static labels are always correct since these apps can not perform malicious behaviors.

While for the samples with "malware" static labels, the collected behaviors from malware apps may not contain any malicious behavior due to the imperfect trigger procedure or the integrated technique designed to bypass the dynamic analysis. In this case, since the dynamic labels of the apps' feature vectors are assigned the same as their static labels, those feature vectors are mislabelled and

training models with such feature vectors and labels kind will surely distort the training of classification model.

Note that we also assume that the triggering module is at least reliable to some extent, which means some of the feature vectors have both their static labels and dynamic labels being correct. However, the ratio of the noisy dynamic labels remains unknown in the approach.

5.3 Dynamic Attention Heuristic

Our approach relies on a new heuristic, named *dynamic attention heuristic*, which stats that the variance of self-attention weights for all values in a feature vector can be used to verify the correctness of this vector's dynamic label, when the vector is generated in dynamic-based Android malware detection approaches.

The heuristic is enlightened by two observations: First, as described above, the attention mechanism can find out the most related values in a feature vector to specific values in the output vector, which has been verified useful in many natural language process (NLP) or Pattern Recognition (PR) approaches for a long time. In this case, we assume it will work well on identifying the most important values from the feature vectors which make the classification model output the corresponding labels.

Secondly, the fragments in the feature vectors representing malicious behaviors are only small parts in the triggered behavior records. Intuitively, malware developers will always try to hide the malicious codes under normal ones, and large sections of malicious codes will be found easily. So they will shorten the malicious codes as much as possible. Furthermore, the behavior records will usually be of great length, due to that the triggering module is designed to cover most of the codes in the app.

In this case, if the static label of an app sample is "malware" and its

malicious behaviors are correctly triggered and recorded. Thus in a well-trained neural network model with attention mechanism, the values in the feature vector corresponding to the malicious behaviors will be assigned with high attention weights since they contribute most to its label, while the rest will be given low weights.

On the opposite, if the static label of an app sample is "malware" and its malicious behaviors are not triggered during the dynamic analysis, the dynamic label of the feature vector is actually mislabelled. In this case, the malicious label is not be contributed by any specific fragments of the feature vector. Thus all these values in the feature vector will be given similar weights by the attention mechanism, and the variances of attention weights in such feature vectors will be smaller compared to the one mentioned above.

5.4 The framework of our Approach

Given a noisy dataset, our approach trains an Android malware detection models within three steps: "Pre-processing," "Noise tolerant model training," and "Malware detection ."

5.4.1 Step I: Pre-processing

We propose the dynamic based analysis by recording the sequences of triggered APIs. In detail, the feature set is produced by Chinese Academy of Sciences and includes 378 APIs, while two open source Android analysis tools "AndroidViewClient" and "Androwarn" are utilized in the API triggering. Each app is installed on a Huawei Mate 20 mobile phone, and scripts built by these two analysis tools are also executed on the device to monitor whether the APIs from the feature set are triggered. When the scripts stop, the collected APIs which are in the list are recorded in a sequence, which is directly used as the feature vector of the target app. Finally, the dynamic label of the generated

feature vector is assigned the same as the static label of the app.

5.4.2 Step II: Noise tolerant model training

Structure of Dynamic Attention Model

The architecture of Dynamic Attention model consists of 4 different layers as follows:

Embedding Layer. After the feature vectors produced in the Pre-processing step, these vectors, which represent the collected sequences of APIs, are then fed into the embedding layer to transform each API into dense vectors of the fixed-length that will be used in the following training. In practice, we use the functions provided by the Keras [3] deep learning framework for the transformation, and the length of dense vectors are set to 200.

Bi-LSTM layer. The second detection layer in the Dynamic Attention model is a Bi-directional Long Short Term Memory(Bi-LSTM) layer which processes the input feature vectors in both forward and backward directions. During the process, each LSTM hidden layer produces a "Hidden Vector Sequence" with the length of 32. The previous LSTM hidden layer's output will be taken by the following LSTM layer as the input. In this case, the LSTM layers can build up historical information among all the values in the input vectors. Within the following updating iterations, the model is able to learn the APIs' semantic at the level of applications. The output of the model, which is in the form of its final hidden vector sequence, is then ready to be fed into the attention following attention layer.

Attention Layer. An attention layer is added to our Dynamic Attention model after the Bi-LSTM layer. This layer is used to figure out the contribution of each value in the vector to the final outputs. With this layer, the contribution of each value in the input vector to its corresponding label can be calculated in the forms of attention scores which are positive values. These scores are used in our

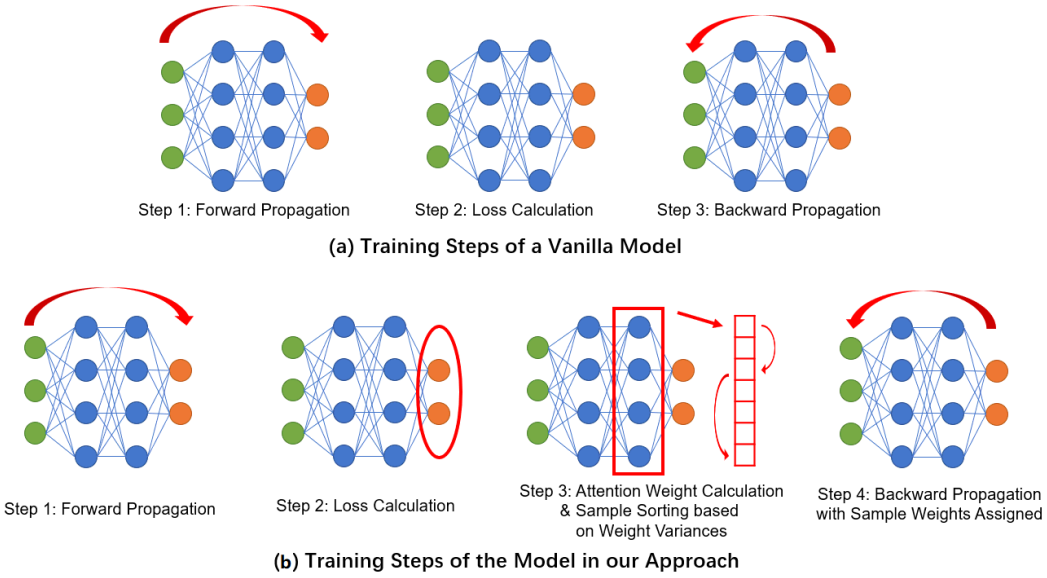


Figure 5.1: Training Dynamic Attention Model with a Batch of Dataset

approach to quantify the contribution to the "malware" label from each API in the form of real values.

Output Layer. A fully-connected layer is added at the end of our classification model which transforms the output of the attention layer to the results of the whole model. The output represents the dynamic label predicted corresponding to the input API feature vector.

Training of Our Detection Model

To build our noise-tolerant model, the training of our classification model is slightly different from the process of training a vanilla model as described in section 5.2.2. In figure 5.1, it is shown in the training of each batch in the forward propagation phase, all the calculation of our approach is the same as that in a vanilla network training. While before the backward propagation phase phase, we insert an additional phase which is named "variance filtering".

While in this additional phase, we first filter out all the input vectors with "malicious" dynamic labels, then we randomly split them into two sub-batch of the same size. The first sub-batch is named "sorting sub-batch", while samples

in this set will be assigned with different sample weights in the next backward propagation phase. The other batch is named “normal sub-batch”.

On calculating the sample weights of vectors in the sorting sub-batch, for each input vector, the variance of attention weights from all values inside the feature vector to the first value in the corresponding output label(In which 1 refers to “malware” and 0 refers to “benignware”) is calculated, the variance is denoted as its “feature variance”. After that, all the inputs are sorted by their corresponding feature variances in ascending order, and each input will then be assigned with a weight which equals to its rank percentage in the sorted order. For instance, if an input’s feature variance is larger than 30% of those in the sub-batch, then it will be assigned with the sample weight of 0.3 in the backward propagation. While for all the other samples including the ones in sorting-batch with “benign” dynamic labels and the ones in the normal sub-batch, all of their sample weights are set to 1.

According to our heuristic, the input feature vectors that have low feature variances will be regarded as their dynamic labels being mistaken and malicious behaviors are not correctly triggered during the preprocessing. Since we assign lower sample weights for them, they have smaller impacts on the model parameter updating during the backward propagation, and the classification model will thus learn less from these samples whose labels are considered to be mistaken.

5.4.3 Step III: Classification

Given the testing set, our approach generates a feature vector for each app in it as shown in Step I: Pre-processing. The trained classification model is then used to output a predicted label for each input feature vector fed to the model.

5.5 The Evaluation

5.5.1 Dataset

Our approach is evaluated on samples that are randomly collected from the App markets and public malware sharing projects. The labels of those apps are further examined by Virustotal to guarantee their correctness. Finally, a dataset consists of 24,092 benignware and 35,693 malware are used in the evaluation.

Feature Vector Generation. As described in section 5.4.1, we extracted the feature vectors from the apps by running them on a Huawei Mate 20 Android mobile phone with the tools "AndroidViewClient" and "Androwarn". After the extraction, we set the dynamic labels for these collected API sequence vectors the same as their static labels.

Generation of Training and Testing Sets. In the introduction section, it is demonstrated that although the correctness of the sample apps' static labels can be guaranteed by downloading the reports from VirusTotal, the dynamic labels for the extracted feature vectors are still questionable as described in the introduction.

To generate a ground truth testing set that can be used to correctly evaluate our Dynamic Attention model, we process the collected dataset with the following steps.

- Intuitively, if an API is included in the feature vector extracted from an app, it is obvious that the API exists in the code of the app. Thus, each app can be transformed into an API set containing all the APIs that appear in its feature vector.

- Two published static analysis-based Android malware detection approaches, SDAC [88] and Drebin [19], are utilized to help build the ground truth testing set. Since both of their feature sets can be generated by the API existence, thus we can then convert the generated API sets

Table 5.1: Overview of Training and Testing Sets

	# of Samples w/ "Malicious" Labels	# of Samples w/ "Benign" Labels
Training Set	33,588	21,987
Testing Set	2,105	2,105

from the last step to feature vectors that can be used by these two static Android malware detection approaches.

- With these compatible feature vectors converted, we applied the trained detection models of SDAC and Drebin on these feature vectors and collect the predicted labels. Given the predicted results generated by these two approaches, we generated the testing set according to the following two criteria: For each sample in the dataset, if (1) its static label is "malware" and (2) its predicted labels from both static approaches are "malware", it is then labelled as "malware" and chosen to build the testing set used in the evaluation of Dynamic Attention.

  Note that the classification results in this step may have a low recall rate on the malware, but its accuracy must be very high. Since the selected ones have only parts of their codes tested by the two approaches, while only these parts of codes have shown enough evidence to be classified as "malware".

- After malware are selected to form the testing set in the last step, the same number of samples having "benign" static labels are randomly chosen from the original dataset and used to form the testing set. In this case, all the dynamic labels for the samples are guaranteed to be correct. And all the unselected ones are then used to build the training set in our approach. The overview of our training and testing sets is shown in table 5.1.

5.5.2 Proof Experiments for our Heuristic

To further demonstrate the heuristic proposed in section 5.3, some experiments are performed in this section to evaluate the key idea of our heuristic: "Samples with mistaken dynamic labels will have a lower feature variance than those with correct dynamic labels".

With the training set generated in the last section, we first train a vanilla classification model with all the samples and labels in the training set. The vanilla model has the same architecture as that in our approach, however, during its training, no weights assigning is performed, which means the sample weights of all the samples are set to 1 in training this model.

Based on this model, the distribution of normalized feature variances in the two following datasets are shown in figure 5.2. In detail, figure 5.2(a) shows the distribution of that in the testing set whose labels are guaranteed to be correct, while figure 5.2(b) shows the distribution of variances for all the feature vectors in the training set whose labels are noisy.

As described in the heuristics, mistakenly-labelled samples will have low feature variances, while correctly-labelled ones have higher variances. Since label noises exist in the training set, feature variances from it will include more low values than those from the testing set, and a similar trend can also be observed from these figures.

5.5.3 Performance Evaluation

To evaluate how much our approach contributes in building the noise tolerant model, we compare the detection accuracy of our approach with that of the vanilla model. The vanilla model does not have its sample weights adjusted in its whole training process, while in our approach, the sample weights are adjusted with the help of the attention mechanism to reduce the impact caused by mistaken dynamic labels. All the other structures and parameters are set to

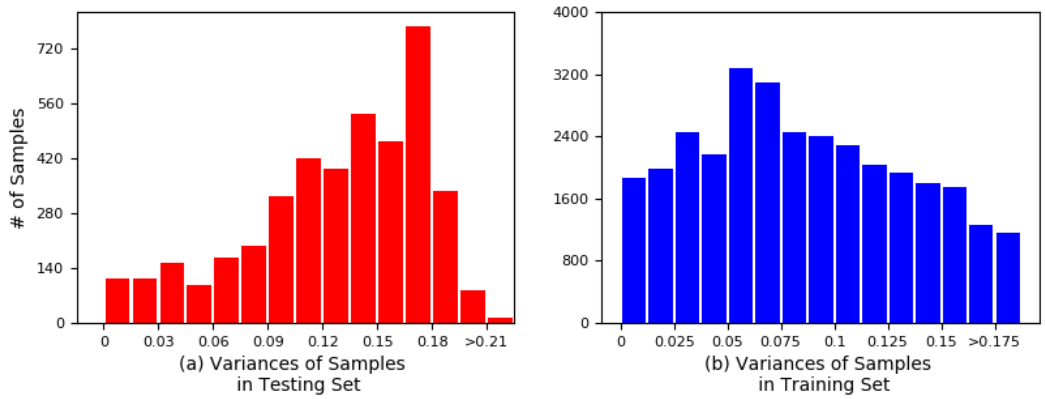


Figure 5.2: Distribution of Attention Weight Variances in Testing/Training Datasets

Table 5.2: Detection Accuracy of our Approach compared with a Vanilla Model of the Same Structure

Model	Detection Performance				
	TP	FN	TN	FP	F-score
Our Approach	1874	231	1521	584	82.13%
Vanilla Model	1428	677	1108	997	63.05%

the same in two models for a fair comparison.

Table 5.2 shows the detection accuracy in form of F-score in these two models. From which it is obvious that our approach is more resistant against the label noises than a vanilla model without using sample weight adjustment. In detail, our approach has more TP and TN results than those in the vanilla model, which demonstrate a higher accuracy in the noise detection.

Chapter 6

Integration of the Three Works

In this chapter, we discussed how the works in this dissertation can be integrated to make Android malware detection more robust. In detail, the three proposed works improve the robustness of Android malware detection in two main aspects: (i) SDAC, as a static-based solution, provides robustness towards the evolvement of Android specifications. (ii) Differential Training provides robustness against label noises as a general framework, while it requires specific Android malware detection solutions for noise reduction. Dynamic Attention provides robustness against noises in the form of a dynamic-based Android malware detection solution, while the noises are caused by the imperfect trigger modules.

Note that within the three works, both SDAC and Dynamic Attention are individual Malware detection solutions, while Differential Training is a framework that needs other solutions to work together. Thus, we can integrate these works as shown in figure 6.1. In the integrated solution, apps will be detected by both the static-based solution SDAC and the dynamic-based solution Dynamic Attention parallelly: The SDAC solution works with the Differential Training framework to provide the robustness towards the evolvement of Android specifications and the label noises simultaneously, which is in the form of static-based analysis. While Dynamic Attention provides the robustness

towards the label noise in the form of dynamic analysis. Then the final prediction results are generated based on predictions from each solution, by calculating the sum of prediction weights for each label. In this case, the integrated solution is able to make the Android malware detection solution more robust in both aspects as listed above.

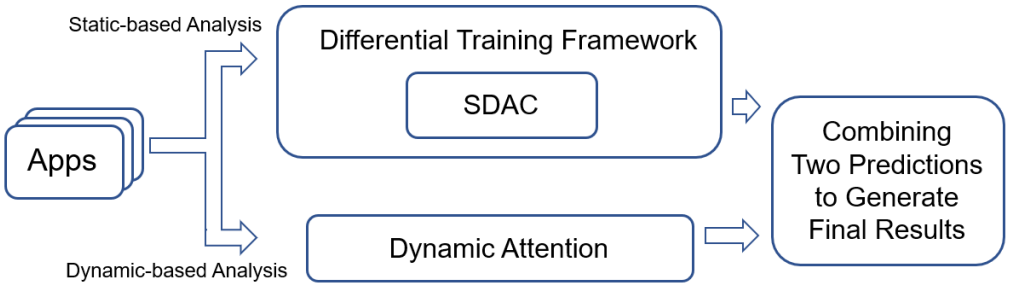


Figure 6.1: Integration of Works

Note that all our three works are implemented on PC platforms, so the integrated solution can also provide the robust Android malware detection service in PC-based scenarios such as app markets. Besides, all the hyperparameters in these three works are either fixed (e.g., the type and structure of classification models in Differential Training and Dynamic Attention) or automatically calculated (e.g., the threshold τ as described in chapter 3.4). Thus, no manual efforts are needed and the integrated solution is fully automated once being deployed.

Chapter 7

Conclusion

This dissertation makes contributions to the robustness of Android malware detection, and it provides three different approaches to achieve its goal.

Our first work is a novel slowing aging solution named SDAC to make Android malware detection robust against changes in Android specifications. The key drivers to achieve slow aging in SDAC include (i) clustering APIs based on the semantic distances among APIs, (ii) evaluating a new API's contribution to malware detection using existing APIs' based on API clusters, and (iii) updating API clusters and classification models based on both training data with true labels and testing data with pseudo-labels. The best versions of SDAC achieve both high accuracy with average F-score 97.49%, and slow aging speed with average F-score decline 0.11% per year over five years in our experiments. The other versions have lower requirements on computing resources, but still perform better than the state of the art.

In the second work, we proposed Differential Training as a generic framework to detect and reduce label noises from training data to make any machine learning based Android malware detection robust against label noises in static analysis. Differential Training is novel due to (i) the use of intermediate states of input samples in the whole training process for noise detection, (ii) the use of downsampled set to maximize the differences between wrongly-labeled

samples and correctly-labeled samples, and (iii) the use of unsupervised outlier detection algorithms for not relying on even a small set of correctly-labeled training samples. Our experimental results show that Differential Training reduces 87.4%, 82.6% and 64.7% wrong labels in the training sets of SDAC, Drebin and DeepRefiner, respectively in the default setting where the noise ratio is set to 10%. With label noises being reduced, the F-scores of these malware detection approaches increase from 89.04%, 73.20% and 91.37% to 97.19%, 84.40% and 93.41%, respectively. The improved F-scores are close to their upper bounds (97.71%, 93.34% and 93.59%). Our experiments also show that the performance of Differential Training is consistent for processing datasets at various noise ratios, and it is superior to the state-of-the-art unsupervised algorithm Co-Teaching for robust training of deep neural networks with noisy labels.

In the third work, we proposed a dynamic-based Android malware detection approach named Dynamic Attention, which is robust against label noises in dynamic analysis of Android malware caused by imperfect trigger procedure. By introducing (i) attention weights of sample feature vectors in the malicious behavior identification and (ii) weighting training samples based on the variances among the attention weights each sample feature vector, we succeed in reducing the negative impact to the classification model caused by dynamic label noise. Our experiment results show that, with the sample weighting based on the feature variance, our approach can increase the detection performance from 63.05% (tested by a vanilla model) to 82.13% in form of F-score.

List of Publications

Conference Papers

J. Xu, Y. Li and R. H. Deng. Differential Training: A Generic Framework to Reduce Label Noises for Android Malware Detection. In *The Network and Distributed System Security Symposium (NDSS 2021)*, Accepted in Dec. 2020.

K. Xu, Y. Li, R. H. Deng, K. Chen, **J. Xu**. Droidevolver: Self-evolving android malware detection system. In *IEEE European Symposium on Security and Privacy (EuroS&P 2019)*, pp.47-62

D. Wu, X. Liu, **J. Xu**, D. Lo and D. Gao. Measuring the declared SDK versions and their consistency with API calls in Android apps In *International Conference on Wireless Algorithms, Systems, and Applications (WASA 2017)*, pp. 678-690.

Journal Papers

J. Xu, Y. Li, R. H. Deng and K. Xu. SDAC: A Slow-Aging Solution for Android Malware Detection Using Semantic Distance Based API Clustering. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, Accepted in June. 2020.

Bibliography

[1] "android developer documentation". https://developer.android.com/reference /classes.

[2] "dexProtector". https://dexprotector.com/.

[3] Keras. https://www.tensorflow.org/api_docs/python/ tf/keras/.

[4] Neurips'18: Co-teaching: Robust training of deep neural networks with extremely noisy labels. https://github.com/bhanML/Co-teaching.

[5] NIPS 2017: Decoupling "when to update" from "how to update". https://github.com /emalach/UpdateByDisagreement.

[6] PyOD. https://pyod.readthedocs.io/en/latest/.

[7] "Scikit-learn". https://scikit-learn.org/stable/modules/generated/sklearn.covariance.EllipticEnvelope.html.

[8] "TensorFlow". https://www.tensorflow.org/api_docs/python/tf/keras/ callbacks/EarlyStopping.

[9] "Wasserstein". https://en.wikipedia.org/wiki/Wasserstein_metric.

[10] "VirusTotal". https://www.virustotal.com/, 2004.

[11] "VirusShare project". https://virusshare.com/, 2011.

[12] Combating potentially harmful applications with machine learning at google: Datasets and models. , https://android-developers.googleblog.com/2018/11/combating-potentially-harmful.html, 2018.

[13] Y. Aafer, W. Du, and H. Yin. "DroidAPIMiner: Mining API-level features for robust malware detection in Android". In *EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*. Springer, 2013.

[14] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino. "Going Native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy.". In *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[15] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee. "Improving accuracy of Android malware detection with lightweight contextual awareness". In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2018.

[16] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. "Androzoo: Collecting millions of android apps for the research community". In *IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016.

[17] A. Apvrille and R. Nigam. "Obfuscation in Android Malware, and How to Fight Back". *Virus Bulletin*, pages 1–10, 2014.

[18] S. Arora, S. S. Du, W. Hu, Z. Li, and R. Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In *International Conference on Machine Learning (ICML)*, 2019.

[19] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. "DREBIN: Effective and explainable detection of Android malware in your pocket.". In *The Network and Distributed System Security Symposium (NDSS)*, 2014.

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[21] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. "Mining apps for abnormal usage of sensitive data". In *International Conference on Software engineering (ICSE)*. IEEE, 2015.

[22] F. A. Breve, L. Zhao, and M. G. Quiles. Semi-supervised learning from imperfect data through particle cooperation and competition. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2010.

[23] G. Brito, A. Hora, M. T. Valente, and R. Robbes. "Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems". In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 1, pages 360–369. IEEE, 2016.

[24] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. "Mast: Triage for market-scale mobile malware analysis". In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2013.

[25] H.-S. Chang, E. Learned-Miller, and A. McCallum. "active bias: Training more accurate neural networks by emphasizing high variance samples". In *Advances in Neural Information Processing Systems (AIPS)*, 2017.

[26] L. Chen, M. Zhang, C.-y. Yang, and R. Sahita. "Semi-supervised classification for dynamic Android malware detection". In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[27] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. "Stormdroid: A streaminglized machine learning-based system for detecting Android malware". In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2016.

[28] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. "Networkprofiler: Towards automatic fingerprinting of Android apps". In *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2013.

[29] F. Dernoncourt, J. Y. Lee, O. Uzuner, and P. Szolovits. "De-identification of patient notes with recurrent neural networks". *Journal of the American Medical Informatics Association (JAMIA)*, 2017.

[30] A. Desnos. "Androguard: Reverse engineering, malware and goodware analysis of Android applications... and more (ninja!)". http://code.google.com/p/androguard, 2015.

[31] S. S. Du, X. Zhai, B. Poczos, and A. Singh. Gradient descent provably optimizes over-parameterized neural networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[32] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang. "Things you may not know about Android (Un) Packers: A systematic study based on whole-system emulation". In *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[33] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". *ACM Transactions on Computer Systems (TOCS)*, 2014.

[34] W. Enck, M. Ongtang, and P. McDaniel. "On lightweight mobile phone application certification". In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2009.

[35] A. Z. Erich Schubert. "ELKI data mining". https://elki-project.github.io/, 2016.

[36] Y. Feng, S. Anand, I. Dillig, and A. Aiken. "Apposcopy: Semantics-based detection of Android malware through static analysis". In *International Conference on Software engineering (ICSE)*. ACM, 2014.

[37] B. Frénay and M. Verleysen. Classification in the presence of label noise: a survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2013.

[38] J. Goldberger and E. Ben-Reuven. Training deep neural-networks using a noise adaptation layer. In *5th International Conference on Learning Representations, (ICLR)*, 2017.

[39] B. Han, Q. Yao, X. Yu, G. Niu, M. Xu, W. Hu, I. Tsang, and M. Sugiyama. "Co-teaching: Robust training of deep neural networks with extremely noisy labels". In *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[40] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. "Hindroid: An intelligent Android malware detection system based on structured heterogeneous information network". In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2017.

[41] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan. "PIndroid: A novel Android malware detection system using ensemble learning methods". *Computers & Security*, 68:36–46, 2017.

[42] L. Jiang, Z. Zhou, T. Leung, L.-J. Li, and L. Fei-Fei. "MentorNet: Learning data-driven curriculum for very deep neural networks on corrupted labels". In *International Conference on Machine Learning (ICML)*, 2018.

[43] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar. "Better malware ground truth: Techniques for weighting anti-virus vendor labels". In *8th ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2015.

[44] "KasperskyLab". "What is Riskware?". https://www.kaspersky.com/resource-center/threats/riskware, 2017.

[45] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im. "A multimodal deep learning method for Android malware detection using various features". *IEEE Transactions on Information Forensics and Security (TIFS)*, 2018.

[46] P. Lantz. "Droidbox - Android application sandbox". https://github.com/pjlantz/droidbox, 2014.

[47] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. "Understanding Android app piggybacking: A systematic study of malicious code grafting". *IEEE Transactions on Information Forensics and Security (TIFS)*, 2017.

[48] Y. Li, J. Yang, Y. Song, L. Cao, J. Luo, and L.-J. Li. "Learning from noisy labels with distillation". In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[49] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. "API change and fault proneness: A threat to the success of Android apps". In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2013.

[50] M. Lindorfer, M. Neugschwandtner, and C. Platzer. "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis". In *IEEE Computer Software and Applications Conference (COMPSAC)*. IEEE, 2015.

[51] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. "Andrubis–1,000,000 Apps Later: A View on Current Android Malware Behaviors". In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2014.

[52] Mariconti. "MaMaDroid project". https://bitbucket.org/gianluca_students/mamadroid_code, 2017.

[53] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. "MaMaDroid: Detecting Android malware by building Markov Chains of behavioral models". In *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[54] T. McDonnell, B. Ray, and M. Kim. "An empirical study of API stability and adoption in the android ecosystem". In *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013.

[55] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.

[56] A. Menon, B. Van Rooyen, C. S. Ong, and B. Williamson. "Learning from corrupted binary labels via class-probability estimation". In *International Conference on Machine Learning (ICML)*, 2015.

[57] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. "Distributed representations of words and phrases and their compositionality". In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[58] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[59] A. Narayanan, L. Yang, L. Chen, and L. Jinliang. "Adaptive and scalable android malware detection through online learning". In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016.

[60] N. Natarajan, I. S. Dhillon, P. K. Ravikumar, and A. Tewari. Learning with noisy labels. In *Advances in neural information processing systems (NIPS)*, 2013.

[61] P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen. "A pragmatic Android malware detection procedure". *Computers & Security*, 70:689–701, 2017.

[62] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin. "Dark Hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps.". In *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[63] G. Patrini, A. Rozza, A. Krishna Menon, R. Nock, and L. Qu. Making deep neural networks robust to label noise: A loss correction approach. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[64] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. "TESSERACT: Eliminating experimental bias in malware classification across space and time". *arXiv*, 2018.

[65] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, pages 1–6, 2014.

[66] V. Rastogi, Y. Chen, and X. Jiang. "Droidchameleon: evaluating Android anti-malware against transformation attacks". In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2013.

[67] R. Řehůřek and P. Sojka. "Software framework for topic modelling with large corpora". In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, 2010.

[68] M. Rhode, P. Burnap, and K. Jones. Early-stage malware prediction using recurrent neural networks. *computers & security*, 2018.

[69] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara. "Experimental study with real-world data for Android app security analysis using machine learning". In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[70] A. I. Schein and L. H. Ungar. Active learning for logistic regression: an evaluation. *Machine Learning*, 2007.

[71] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim. "FLEXDROID: Enforcing in-app privilege separation in Android.". In *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[72] B. Settles. "Active learning literature survey". Technical report, 2009.

[73] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting". *The Journal of Machine Learning Research (JMLR)*, 2014.

[74] S. Sukhbaatar, J. Bruna, M. Paluri, L. Bourdev, and R. Fergus. Training convolutional networks with noisy labels. *arXiv preprint arXiv:1406.2080*, 2014.

[75] "Trendmicro". "A Case of Misplaced Trust: How a Third-Party App Store Abuses Apple's Developer Enterprise Program to Serve Adware". https://blog.trendmicro.com/trendlabs-security-intelligence/how-a-third-party-app-store-abuses-apples-developer-enterprise-program-to-serve-adware/, 2016.

[76] R. Tsuchida, F. Roosta, and M. Gallagher. Invariance of weight distributions in rectified MLPs. In *International Conference on Machine Learning (PMLR)*, 2018.

[77] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

[78] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.

[79] N. Viennot, E. Garcia, and J. Nieh. "A measurement study of Google play". In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233. ACM, 2014.

[80] T. L. Wang. "AI-Based Antivirus: Detecting Android malware variants with a deep learning system". In *Blackhat Europe*, 2016.

[81] X. Wang, S. Zhu, D. Zhou, and Y. Yang. "Droid-AntiRM: Taming control flow anti-analysis to support automated dynamic analysis of Android malware". In *Proceedings of the 33th Annual Computer Security Applications Conference (ACSAC)*, 2017.

[82] Y. Wang, W. Liu, X. Ma, J. Bailey, H. Zha, L. Song, and S.-T. Xia. "Iterative learning with open-set noisy labels". In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[83] F. Wei, O. X. Roy, Sankardas, and Robby. "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 (CCS)*. ACM, 2014.

[84] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[85] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao. "Measuring the declared SDK versions and their consistency with API calls in android apps". In *International Conference on Wireless Algorithms, Systems, and Applications (WASA)*. Springer, 2017.

[86] Y. Wu, S. C. Hoi, C. Liu, J. Lu, D. Sahoo, and N. Yu. "Sol: A Library for Scalable Online Learning Algorithms". *Neurocomputing*, 260:9–12, 2017.

[87] B. Xie, Y. Liang, and L. Song. Diverse neural network learns true target functions. In *Artificial Intelligence and Statistics (PMLR)*, 2017.

[88] J. Xu, Y. Li, R. Deng, and K. Xu. SDAC: A Slow-aging solution for Android malware detection using semantic distance based API clustering. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2020.

[89] K. Xu, Y. Li, and R. H. Deng. "ICCDetector: ICC-based malware detection on Android". *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(6):1252–1264, 2016.

[90] K. Xu, Y. Li, R. H. Deng, and K. Chen. Deeprefiner: Multi-layer Android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018.

[91] L.-K. Yan and H. Yin. "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis.". In *USENIX security symposium*, 2012.

[92] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications". In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2014.

[93] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. "Appcontext: Differentiating malicious and benign mobile app behaviors using context". In *International Conference on Software engineering (ICSE)*. IEEE, 2015.

[94] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. "Appspear: Bytecode decrypting and dex reassembling for packed Android malware". In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2015.

[95] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection". In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2013.

[96] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. "Droid-sec: deep learning in Android malware detection". In *ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[97] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. "Semantics-aware Android malware classification using weighted contextual API dependency graphs". In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[98] Z. Zhang and M. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. In *Advances in neural information processing systems (NIPS)*, pages 8778–8788, 2018.

[99] Z. Zhu and T. Dumitras. “Featuresmith: Automatically engineering features for malware detection by mining the security literature”. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.