University of Technology Sydney

Faculty of Engineering and Information Technology

Machine Learning Detection and Analysis on Obfuscated Android Malware

by

Yanxin Zhang

A Thesis Submitted

for the Degree of

Doctor of Philosophy

Sydney, Australia

January 20, 2022

Certificate of Authorship/Originality

I, Yanxin Zhang declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Computer Science, Faculty of Engineering and IT at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Production Note:
Signature: Signature removed prior to publication.

Date: January 20, 2022

Abstract

In recent years, the Android platform is the prevalent operating system platform, and it accounts for more than half of the world's mobile operating system market share. With the popularity of Android smartphones and Android tablets, Android-based malware has also developed rapidly, so malicious code detection technology based on the Android platform has been proposed, and Android application security must be carefully studied and have an extensive discussion. Meanwhile, machine learning methods can simulate the behavior of Android applications and distinguish between benign and malicious software; therefore, it is becoming the key to the effective detection of Android malware. However, malware developers may understand detection systems and take countermeasures against detection methods in the real world. Most current research work ignores this problem.

Recent research in machine learning has shown that learning-based systems are susceptible to well-designed adversarial examples. For instance, malware detection based on static analysis will encounter the challenge of code obfuscation, which is the countermeasure for malware authors to bypass detection. Therefore, there is a strong demand for research on Android malware detection from the perspective of defenders and attackers.

My research direction is to explore machine learning methods to ensure the security and stability of Android applications. Malware writers often use obfuscation techniques to obfuscate Android malware to evade detection by Android malware detectors. Knowing about the impact of code obfuscation on Android malware can provide valuable insights into obfuscated malware and, therefore, the design of resilient Android malware detectors.

First, labeling malware or malware clustering is essential for identifying new security threats, triaging, and building reference datasets. The state-of-the-art Android malware clustering approaches rely heavily on the raw labels from commercial AntiVirus (AV) vendors, which causes misclustering for a substantial number of

weakly-labeled malware due to the inconsistent, incomplete, and overly generic labels reported by these closed-source AV engines, whose capabilities vary greatly and whose internal mechanisms are opaque (i.e., intermediate detection results are unavailable for clustering). The raw labels are thus often used as the only significant source of information for clustering. To address the limitations of the existing approaches, we present ANDRE, a new Android Hybrid Representation Learning approach to clustering weakly-labeled Android malware by preserving heterogeneous information from multiple sources (including the results of static code analysis, the meta-information of an app, and the raw-labels of the AV vendors) to learn a hybrid representation for accurate clustering jointly. The learned representation is then fed into our outlier-aware clustering to partition the weakly-labeled malware into known and unknown families. The malware whose malicious behaviors are close to those of the existing families on the network is classified using a three-layer Deep Neural Network (DNN). The unknown malware is clustered using a standard density-based clustering algorithm. The evaluation shows that ANDRE effectively clusters weakly-labeled malware that cannot be clustered by the state-of-the-art approaches, achieving comparable accuracy with those approaches for clustering ground-truth samples.

Second, Android piggybacked malware (i.e., apps that piggyback malicious code) are becoming ubiquitous in app stores. Malware writers often use obfuscation techniques to obfuscate piggybacked apps to evade detection by Android malware detectors. Previous studies in this field have focused on the impact of code obfuscations on the detection of piggybacked malware, but the impact of code deobfuscation on detecting obfuscated piggybacked apps has rarely been investigated. Knowing about the impact of code deobfuscation can provide helpful insights into obfuscated piggybacked apps and, therefore, the design of resilient Android malware detectors. We conduct an empirical study of code deobfuscations on detecting obfuscated Android piggybacked apps, focusing on three types of malware detectors: commercial anti-malware products, machine learning-based detectors, and similarity-based detectors. We observe that code deobfuscations can have various impact on the detection rate for different types of malware detectors, such as similarity-based or machine learning-based detectors. Also, we observe that the examined deobfuscation tools have a different impact on obfuscated piggybacked apps after deobfuscations.

Thrid, Android malware adversarial samples can help test the weaknesses of antivirus products to improve their robustness against emerging and sophisticated Android malware. A typical method to generate testing samples is to transform a single Android malware into a range of samples through code obfuscation. Existing obfuscation tools often have many obfuscation options, which can generate a large number of testing samples. However, accurately identifying top evasive samples to gain an optimal cost-benefit trade-off is a challenge. To address the above issue in adversarial malware generation for robustness testing of antivirus products, we present ALBS(Active Learning-based Sampling), an active-learning-based approach, by constructing an adversarial ranking model to select the most powerful evasive adversarial samples. First, ALBS generates all Android malware adversarial samples and then extracts the adversarial samples' representation using both unstructured and structural code features. Then, it utilizes the multi-view multi-learner (MVML) active learning method to train a model which ranks the adversarial samples using multiple learning-to-rank (LTR) algorithms. Our empirical experiments demonstrate the effectiveness of ALBS for generating adversarial samples for antivirus product robustness testing. The experiment result indicates that the proposed active learning approach can optimize the adversarial sample generation process and effectively select the most potent testing samples, improving the efficiency of Android antivirus products' robustness testing.

Acknowledgements

I want to express my deep gratitude to all those who have given me help and support in writing my thesis. First of all, I would like to thank my supervisors, Dr. Yulei Sui and Prof. Ivor Tsang, for their guidance, support, and advice during my PhD study. They discovered my potential, encouraged me to go further, and provided precious lessons in many different fields. Their meticulous academic attitude and dedicated work style have always been my role models. Also, I am grateful to the University of Technology Sydney for providing generous supports and attractive academic environment during my candidature career.

I would also like to thank the colleges in my group for discussing research work with me and sharing their knowledge and suggestions generously. They provided many insightful discussions for my research and offered various help to my personal life.

Finally, I would like to thank my parents and wife for their love, care, and continuous support and encouragement. Without their valuable support, it would be impossible for me to finish this thesis.

Sydney, Australia, January 20, 2022

List of Publications

Below is a list of publications that are included in this thesis as Chapter 3, 4, 5 respectively.

Chapter 3:

Zhang, Y., Sui, Y., Pan, S., Zheng, Z., Ning, B., Tsang, I. and Zhou, W., 2019. "Familial clustering for weakly-labeled android malware using hybrid representation learning." *IEEE Transactions on Information Forensics and Security, 15, pp.3401-3414.* Status: Published

Chapter 4:

Zhang, Y., Xiao, G., Zheng, Z., Zhu, T., Tsang, I.W. and Sui, Y., 2020, December. "An Empirical Study of Code Deobfuscations on Detecting Obfuscated Android Piggybacked Apps." *In 2020 27th Asia-Pacific Software Engineering Conference (APSEC) (pp. 41-50). IEEE.* Status: Published

Chapter 5:

Zhang, Y., Cai, H., Tsang, I.W. and Sui, Y. "ALR: Active Learning Based Ranking for Adversarial Android Malware Generation." *Submitted to IEEE Transactions on Reliability.* Status: Under review

Other publications during candidature:

Zhang, Y.. "Obfuscation Resilient Lightweight Android Repackaged Apps detection. " *Submitted to IEEE Transactions on Dependable and Secure Computing.* Status: Under review

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In this chapter, we briefly introduce our research problems' background and summarize the topics and contributions of our proposed models.

1.1 Background

1.1.1 Weakly-labeled Android Malware Family Clustering

Android malware family. Android malware which belong to same malicious family usually have similar behavior patterns, and newly intercepted Android malware are variants of existing family. For example, the plankton family, whose first family member appeared in 2011, is still one of the most widely spread Android malware [3]. Even in the official Android application store Google Play, the plankton malicious code also appears in a large number of applications. Its family members are mainly different versions of adware. The primary malicious behavior is downloading unwanted advertisements on mobile phones, changing mobile devices' browsing homepage, or adding new shortcuts and bookmarks. In recent years, there have been work discoveries that divide newly detected malicious samples into their corresponding families and selecting representative samples in each family for further detailed analysis by security personnel can greatly reduce the workload of security personnel [4–6], so malicious family identification has also become one of the main research issues in malware detection today.

Weakly-labeled malware. We define two types of weakly-labeled malware based on the results from the state-of-the-art approaches [1, 2] with their statistics given in Tables 3.1 and 3.2: (1) *Malware with empty label.* The tokens from the raw labels being recognized as generic tokens (e.g., `apprisk`) or unable to be parsed by the rules of AVClass and Euphony are discarded, resulting in weakly-labeled apps without a family name. Figure 3.1 illustrates a concrete example of malware with an empty label. In addition, for example, as listed in Table 3.1, AVClass and Euphony are unable to produce a family name for 34.06% and 17.04% of the total 9000 apps which were recently uploaded to VirusTotal between Mar. 2017 and Feb. 2018, resulting in **1534** weakly-labeled malware. (2) *Malware with controversial labels.* The plurality voting strategy is also hard to disambiguate inconsistency between two family names reported by a close number of vendors among the 7466 apps which have family names extracted from the raw labels by Euphony. In Figure 3.2, an example of malware with controversial labels is given. Table 3.2 shows that the top two most frequent names are reported by an equal number of vendors for **1790** apps (24%), causing the plurality voting to be less confident for these weakly-labeled malware compared to their strongly-labeled counterpart with an uncontroversial family name reported by the majority (e.g., 80%) of all available vendors in VirusTotal [7].

Hybrid representation learning. Deep representation learning (DRL) [8] is a new and promising branch of machine learning. DRL learns the representation of the target data via deep architectures in a layer-wise manner. The higher abstraction level of the features is embedded in a lower unified representation, making it easy for later classification and clustering tasks. Recently, Hybrid Representation Learning approaches [9–12] have significantly enhanced the existing DRL techniques in terms of both efficiency and accuracy for training and predicting large-scale networks by extracting heterogeneous information in a variety of formats (e.g., texts [13] or graphs [14]) and from multiple data sources.

1.1.2 Investigating Code Deobfuscations on Detecting Obfuscated Piggybacked Apps

Android code obfuscation. Code obfuscation does not make the code uncompiled. On the contrary, it transforms the code on structured code features (e.g., modifying program's control-flows) or on unstructured code features (e.g., renaming the apps' classes, methods, variables to meaningless names). In this PhD thesis, we use four types of obfuscation strategies, i.e., control flow flattening, insert junk code, identifier obfuscation, and string obfuscation, as shown in Table 4.1.

1) *Control Flow Flattening (CFF).* This type of obfuscation strategy represents the control statements in java code (e.g., "if", "while", "for", and "do"), which are converted into switch branch statements without changing the function of the source code. The advantage of CFF is that it blurs the relationship between the code blocks in the switch, increasing the difficulty of analysis. This technique divides the method into multiple basic blocks (case code blocks) and an entry block and allows these basic blocks to have both a standard predecessor module and successor module to achieve flattening. The predecessor module mainly performs the distribution of basic blocks, and the distribution is realized by changing the switch variable. The successor module is used to update the value of the switch variable and jump to the beginning of the switch.

2) *Insert Junk Code (IJC).* This type of obfuscation strategy inserts a set of useless bytes into an original program without changing the original logic of the program. The program will still run normally, but the disassembly tool will fail to disassemble the inserted bytes, e.g., in a manner that the disassembly tool reports an error when the first few bytes of the typical instruction are recognized as incomplete. Therefore, the inserted instructions are random and incomplete. Note that these instructions must satisfy two conditions. Firstly, the instructions are located in a path that will never be executed when the program is running, and secondly that these instructions are part of the legal instructions except that they are incomplete instructions.

3) *Identifier Obfuscation (IO).* Identifier obfuscation is to rename the packages,

classes, methods, and variables in a source program and then replace them with meaningless identifiers, making it harder to crack the identifiers for analysis.

4) *String Obfuscation (SO).* To avoid the disassembled code being easy to analyze and understood, more critical string variables in the source program are often obfuscated. There are two kinds of string obfuscation strategies, namely encoding obfuscation and string encryption. Encoding obfuscation first converts the string into a hexadecimal array or Unicode encoding and then restores it to a string when the string is called. After encoding obfuscation takes place, a series of numbers or garbled characters, which are difficult to analyze directly, exist in the reverse direction. String encryption is the local encryption of the string and then encode the source code into the ciphertext. Furthermore, implementing a decryption function as well as calling the decryption function decrypts the ciphertext where it can be used.

Android code deobfuscation. Code deobfuscation is a reverse process of code obfuscation. The code deobfuscation usually deobfuscates the obfuscated according to three key aspects, as follows: (1) *readability.* The purpose of deobfuscation is to make the code readable. The simpler the code, the more readable it is; (2) *better understanding of program's control- and data-flows.* This helps us to analyze the possible execution flow of the program statically; and (3) *getting context.* The contextual relevance of the program can help us better understand the semantic of a program. Several attempts working on the code deobfuscation of Android apps have been conducted. Deguard [15] was proposed to deal with layout obfuscation introduced by ProGuard. The key idea is to summarize a probabilistic model by learning unobfuscated apps on a large scale and then use the model to recover the obfuscated code. Also, Baumann *et al.* [16] used a similar approach to perform ProGuard deobfuscation by code matching.

1.1.3 Active Learning Based Ranking for Adversarial Android Malware

Adversarial Android malware. With the increasing requirements for Android malware detection accuracy and performance, more and more Android malware detection engines use artificial intelligence algorithms. At the same time, attackers began to try to make certain modifications to Android malware, so that Android malware can bypass these machine learning-based detections while retaining its own functionality. The above process is an adversarial attack in the field of Android malware detection.

It is worth noting that the concept of adversarial samples first proposed by Goodfellow in the field of image recognition has also been applied to Android malware detection. That is, by adding some disturbances that do not affect the normal operation of the Android application, the algorithm cannot identify the application as malicious. The perturbation is added in the input layer of the malware detection algorithm. For the algorithm whose input is the permission feature, the perturbation is the modification of the permission applied for by the application; for the algorithm whose input is the control flow graph, the perturbation is the modification of the control flow.

Although there are some studies on defense against adversarial attacks to improve model security [17], according to the generation principle of adversarial samples, as long as the model using gradients, whether it is deep learning or machine learning using gradient update, there will be adversarial samples, and adversarial samples will exist at the same time. The generation of the perturbation direction is determined by the gradient, so as long as the artificial intelligence algorithm model can determine the perturbation direction according to the structure of the model or the update method of the model, it faces the risk of adversarial attacks. Therefore, enhancing the anti-adversarial ability of Android malware detection engines based on artificial intelligence algorithms is also urgently needed.

1.2 Research Topics

1.2.1 Weakly-labeled Android Malware Family Clustering

Labeling a malicious app as an unknown or a variant of an existing family is vital for identifying new threats, determining the severity of the threat, creating signatures for malware detection, malware triaging, and building reference datasets. Labeling malware is a non-trivial task. The raw labels reported by AntiVirus (AV) vendors are well-known to be inconsistent (e.g., two vendors may report two aliased family names for the same type of malware. `solimba` and `firseria` are aliases) without a standard naming convention (e.g., the conventions CARO [18] and CME [19] are not often used by AV vendors). Although manual inspection for malware labeling by an expert can provide an accurate solution, it is exceptionally costly in practice due to many apps being released every day.

These *raw-label-based* approaches rely heavily on the original labels from the AV vendors. A substantial number of *weakly-labeled malware* are unable to be clustered because their raw labels are often incomplete, inconsistent, and overly generic, as reported by incompatible commercial vendors with different capabilities in the presence of rapidly evolving malware.

A strongly labeled malware with an exact family name is easy to be clustered. However, relying on vendors' raw labels as the only source of information for labeling weakly-labeled malware is inherently partial and shallow. Apart from the reports from AV vendors, an Android app itself is a comprehensive package containing source code and meta-info (e.g., configuration and resource files), which are crucial for extracting the behaviors of an app. This information cannot be easily obtained and is not often considered by the existing raw-label-based clustering approaches. It is because inferring the malware family by analyzing an app is to develop another AV engine, competing with dozens of available AV vendors, which is complicated and likely to cause inconsistent results due to lack of ground-truth family names and the unknown mechanisms of the commercial AV vendors, whose capabilities and mechanisms vary greatly with no intermediate detection results reflected in their reports. Thus, the raw labels generated from these vendors are the only easy-to-use and essential data source for the existing clustering approaches. Through the API provided by VirusTotal, we can collect these raw labels and treat them as textual data for the training the machine learning models.

1.2.2 Investigating Code Deobfuscations on Detecting Obfuscated Piggybacked Apps

Researchers have developed deobfuscation tools to identify obfuscated codes injected into apps by malware writers to overcome the challenge of detecting obfuscated Android malware. For example, Simplify [20], developed by Caleb Fenton, uses a virtual machine sandbox for executing an app to understand its behavior. Then, it tries to optimize the code so that the decompiled code is simplified and more manageable for humans to understand. Different from the work by Fenton, Bichsel *et al.* [15] developed Deguard, a statistical deobfuscation tool for Android. Deguard phrases the layout deobfuscation problem of Android apps as a structured prediction in a probabilistic graphical model for identifiers based on the occurrence of names.

However, there is little empirical evidence on the impact of code deobfuscations on detecting obfuscated Android piggybacked apps. Such empirical knowledge can provide valuable insights for researchers and security engineers to understand better the influences of obfuscation strategies and deobfuscation tools on Android malware detectors. Thus, such evidence can guide the design of resilient and effective Android malware detectors. Therefore, we perform a large-scale empirical study of the impact of code deobfuscations on detecting obfuscated Android piggybacked apps.

1.2.3 Active Learning Based Ranking for Adversarial Android Malware

A typical method to generate penetration testing samples is to transform a single Android malware into a range of samples through code obfuscation. Existing obfuscation tools often have many obfuscation options, which can generate a large number of testing samples. However, accurately identifying top evasive samples to gain an optimal cost-benefit trade-off is a challenge.

It is impossible to utilize all generated adversarial samples for penetration testing. The executable strategy is to pick up the samples with the most potent evading capability to save penetration testing costs. Hence, it is essential to develop a method to select adversarial samples that have the most potent ability to evade antivirus products at a much lower cost. Ideally, the top evasive samples should be identified accurately before the testing, which requires the approach to achieve comparable precision in ranking the top evasive samples like that of state-of-the-practice antivirus engines.

1.3 Research Contributions

1.3.1 Weakly-labeled Android Malware Family Clustering

The principal contributions of our proposed weakly-labeled Android malware family clustering are summarized as follows:

- We present a new hybrid representation learning approach to cluster weakly-labeled Android malware.

- We propose a new representation by successfully preserving heterogeneous information, including raw labels from AV vendors, meta-info ,and results from source code analysis. This provides a compact and low-dimensional representation for effective Android malware clustering.

- We have conducted a comprehensive evaluation using 5,416 ground-truth samples from the Drebin dataset and 9,000 malware from VirusShare, uploaded between Mar. 2017 and Feb. 2018, consisting of 3324 weakly-labeled malware. The results show that our approach has comparable accuracy to the state-of-the-art approaches in clustering ground-truths and can effectively cluster weakly-label malware, which cannot be clustered by AVClass and Euphony.

1.3.2 Investigating Code Deobfuscations on Detecting Obfuscated Piggybacked Apps

The contributions of investigating the Android code deobfuscation on detecting obfuscated piggybacked apps can be summarized as follows:

- To the best of our knowledge, this is the first work to study the impact of code deobfuscations on detecting obfuscated Android piggybacked apps.
- We have generated 11,378 obfuscated and deobfuscated piggybacked apps from 1,399 original piggybacked apps using 10 strategies.
- We have evaluated the precision of three types of Android malware detectors, including commercial anti-malware products (i.e., VirusTotal), machine learning-based detectors (i.e., Drebin and CSBD), and similarity-based detectors (i.e., Androguard and SimiDroid) on the generated obfuscated and deobfuscated piggybacked apps.

1.3.3 Active Learning Based Ranking for Adversarial Android Malware

The contributions of ranking adversarial malware samples are summarized as follows:

- We adopt the multi-view multi-learner (MVML) active learning method to support constructing the adversarial ranking model. The experiment results indicate that ranking models' performance will improve by applying the active learning method selected training set than using the randomly selected training set.
- We use the ensembling strategy with 6 LTR models, namely MART, RankBoost, AdaRank, Coordinate Ascent, Lambdamart, and Random Forest, to rank the adversarial samples generated from Android malware. Using ALR, we can accurately identify the top evasive samples.

- We conducted a comprehensive evaluation using 100,000 adversarial samples. The results show that applying the MVML active learning method to collect the training set can improve the adversarial samples generation process for penetration testing.

1.4 Thesis Organization

This thesis studies the possibility of improving the reliance and resilience of machine learning methods on detecting obfuscated Android malware. The challenges of using machine learning guided detection technology to be more resilient against the obfuscated Android malware are addressed. This thesis is organized as follows:

- *Chapter 2:* This chapter reviews the state-of-the-art works of machine learning guided Android malware detection and obfuscated Android malware, including Android malware classification and clustering, Android obfuscation strategies, etc.

- *Chapter 3:* This chapter presents a new hybrid representation learning approach to cluster weakly-labeled Android malware. We propose a new representation by successfully preserving heterogeneous information, including raw labels from AV vendors, meta-info, and results from source code analysis. This provides a compact and low-dimensional representation for effective Android malware clustering.

- *Chapter 4:* This chapter presents the first work to study the impact of code deobfuscations on detecting obfuscated Android piggybacked apps.

- *Chapter 5:* This chapter addresses the challenges of accurately identifying top evasive samples to gain an optimal cost-benefit trade-off by adopting the multi-view multi-learner (MVML) active learning method to construct the adversarial ranking model.

- *Chapter 6:* This chapter concludes the thesis and presents the future works.

CHAPTER 2

Literature review

2.1 Weakly-labeled Android Malware Family Clustering

2.1.1 Android Malware Detection

Several solutions to Android malware detection [21–25] exist that gives a binary decision to identify whether or not an app is malicious. DREBIN [22] proposes a malware detection approach through a two-class SVM by performing a lightweight static analysis to extract API calls and manifest files as the input features. MaMaDroid [26] leverages sequences of abstracted method calls to create a probabilistic representation of program behaviors in the form of Markov chains. DroidAPIMiner [21] mines API features for malware detection in Android using a lightweight KNN-based binary classification. HINDroid [27] proposes an Android malware detection approach using a heterogeneous information network. Kim et al. [28] present a deep-learning-based approach to malware detection by utilizing the features extracted from an Android app. Unlike malware detection, malware clustering or labeling (such as ANDRE) performs fine-grained familial identification, which in turn can be used to build references datasets for supervised detection and classification [1].

2.1.2 Android Malware Classification and Clustering

DroidMiner [29] proposes a two-level behavioral graph model and extracts sensitive paths to represent malicious behavioral patterns for malware classification. DroidSIFT [6] classifies Android malware through dependency graphs. The approach excludes common third-party libraries to improve precision by using a set of benign subgraphs to remove the common subgraphs of the whole dependency graph. DroidCat [30] proposes a dynamic malware detection and classification approach that complements existing static program analyses by supporting dynamic features such as reflections and callbacks. SMART [31] presents a malware classification approach based on semantic clones using deterministic symbolic automation.

Malware clustering mainly works in an unsupervised or semi-supervised scenario to cluster unlabeled samples. Most existing works on Android malware clustering rely on the raw labels reported by AntiVirus vendors [22]. However, the lack of common naming standards results in inconsistencies among the reports from different vendors [32]. To address this problem, AVClass [1] processes the raw label from vendors by performing alias detection and generic token removal based on vendor-specific rules to produce a single label per sample. Euphony further enhances its accuracy by using self-defined extraction rules to distinguish fields of a raw label. Consensus between different vendors is then reached through plurality voting. Li et al. [33] present a malicious payload mining method for clustering malware by considering the source code information of an app. However, the approach relies on a pre-labeling phase similar to AVClass, and weakly-labeled samples are not included in their clustering process. This PhD thesis addresses the limitations of previous approaches to clustering weakly-labeled malware using a new representation learning approach that preserves code similarity, meta-info, and vendors' raw labels.

2.1.3 Representation Learning

Representation learning takes samples and their corresponding raw features as input and produces a unified and low-dimensional representation, particularly useful for later machine-learning tasks, such as classification and clustering. Its goal is to learn a good encoder to help map high-dimensional information such as images or graph into a low-dimensional representation space. An excellent representation space should be able to extract high-dimensional semantic information in the original signal.

Many existing methods perform representation learning based on a single source of information, such as natural languages (word2vec [34] and doc2vec [13]) and graph structures (DeepWalk [35], Node2vec [36], and LINE [37]). [38] establishes a contrastive learning method from the perspective of time series signals, and trains the encoder based on maximizing mutual information. [39] proposes the deep InfoMax method, which still uses the idea of contrastive learning, but makes it more refined and considers multiple aspects to train the encoder. Its goal is still to maximize the mutual information between pairs of positive samples.

2.2 Investigating Code Deobfuscations on Detecting Obfuscated Piggybacked Apps

2.2.1 Android Obfuscation Strategies

Some work on Android-specific obfuscation tools has previously been conducted. For example, Rastogi *et al.* [40] presented Droid Chameleon, a tool for obfuscating Android apps. Zheng *et al.* [41] proposed ADAM, a framework for obfuscating Android apps and testing them on anti-malware products. Ghosh *et al.* [42] used a reflection call mechanism in Android application development to hide calls to sensitive APIs and access sensitive data. However, this method can be identified by inspecting whether Java reflection calls are being used. Harrison *et at.* [43] analyzes the code obfuscation technology that resists reverse engineering of Android applications. With this technology, obfuscation is divided into name obfuscation, control flow obfuscation, and instruction encoding obfuscation. Naming obfuscation converts the name of a package, class, or method into a meaningless string. However, since DEX bytecode is always stored in memory in the form of clear text during execution, an attacker can dynamically analyze the name obfuscation and implement the DEX bytecode loading logic [44]. Some other studies also focused on the effects of obfuscations on anti-malware products without proposing new obfuscation tools. Pomilia [45] studied the performance of nine anti-malware products on an obfuscated. Maiorca *et al.* [46] studied the effects of code obfuscated by a single tool on 13 anti-malware products.

Code obfuscation technology primarily obfuscates the program's symbol information, such as modifying variable names, adding invalid code blocks, etc. Table 3.3 shows the obfuscation strategies in detail. So that features based on Dalvik bytecode can be easily modified, thereby increasing the difficulty of detection. To bypass the existing malware detection technology, malicious code creators will continuously modify the malware's source code to create variants. The research in [47] show that 86% of malware is produced by repackaging, that is, by implanting the malicious loading code into the Dalvik code of the original app and then repacking and uploading it to the app market to attract users to download it for free. The producer continually modifies the malicious loading code of the same family. The same malicious behavior will have different code implementations.

The increasingly mature obfuscation technology also makes the obfuscated malicious code have a great change in expression while retaining the normal execution of malicious behavior, making the detection of malware more difficult. Basic obfuscation techniques include layout adjustment, renaming, and control flow obfuscation [48, 49]. Advanced obfuscation techniques include reflection mechanism [50] and reinforcement technology [51] and hide malicious behavior in native code [52–54]. There are also many existing obfuscation tools. Typical tools are DashO [55], ProGuard [56], DexGuard [57], Allatori [58], etc. The research on Android program obfuscation technology is also a trendy topic [59–61].

With the continuous development of malware detection technology, malware developers are continually updating malware countermeasure technologies to extend the survival time of malware and maximize benefits to evade the detection of existing technologies. Based on its implementation principles, we divide the existing countermeasure technologies into two main categories: reverse countermeasure technology and countermeasure sample generation technology.

Some specific methods are proposed for reverse countermeasure technology to make the current malware detection technology more robust. For the static countermeasure technology, Li et al. [50] proposed DroidRA, which can effectively identify the use of reflection technology-sensitive API calls. The RevalDroid proposed by Garcia et al. [62] analyzed native code and used its related function calls as features. The PackerGrind proposed by Xue et al. [51] detected whether the malware was hardened and then unpacking it. For dynamic countermeasures technology, Fratantonio et al. [63] proposed TriggerScope, which used symbolic execution, path prediction, and other techniques to detect specific triggers in malware, including time and location. Although these techniques can be used to a certain extent to enhance malware detection capabilities, new evasion technologies for these technologies will also evolve.

The adversarial sample generation technology generates samples that can deceive the model by looking for the machine learning model's weaknesses. Its earliest application is image classification. By injecting noise into the image, generating adversarial samples that deceive the classifier can make the classifier error, for example, recognizing women as men, etc. Malicious developers then applied this technology to malware classification and modified some features to deceive the already-built classifier while ensuring that the malicious behavior remained unchanged. They were classified into benign samples, thereby bypassing the detection. There are several typical related work [64–66]. The main idea is based on the forward gradient algorithm. The forward gradient is the deviation of the target category output value of the neural network for each input feature. Iteratively modify the feature vector of the input object until the maximum perturbation is reached or a successful adversarial sample is generated.

The typical detection methods for adversarial samples include model distillation technology and retraining technology. Model distillation technology adds the category probability knowledge distilled from the original DNN to a small DNN, thereby improving the model's generality while maintaining the classification accuracy. It can reduce the impact of input disturbances on the classifier model. The retraining technique adds adversarial samples to the model training so that the trained model has stronger robustness. However, the shortcomings of the existing methods are more pronounced. It can effectively detect limited adversarial samples, and the accuracy of newly produced adversarial samples is low.

The highly resilient malware detection problem is a typical adversary environment problem, and it has become a research hotspot today. However, today's detection methods are mainly passive defense. After the attack occurs, corresponding solutions are proposed for specific attack problems. Most detection methods keep following security threats and are always passively beaten. In future research, how to change from passive to active detection and control the cause of attacks is a problem worthy of researchers' exploration.

2.2.2 Android Deobfuscation

Schulz *et al.* [67] proposed a deobfuscation method for the string encryption option provided by DexGuard, an obfuscation tool for Android. DexGuard first extracts the decoded string, and later reverse obfuscation is performed by replacing the encrypted string with the extracted decryption string and deleting the decryption method. However, if multiple class encryption options are applied so that the decryption method cannot be found, there is the limitation that reverse obfuscation cannot be performed. Piao *et al.* [68] also analyzed DexGuard and analyzed string encryption and class encryption options. In the case of the string encryption option, it was shown that the decrypted string is output to the log when the decryption method is called and that the log can be extracted by demonstrating that the log is identical to the original string. The method, key, and initial vector are extracted, and the decrypted class file is extracted through the decryption method call. In the case of APKs, the approach cannot perform the deobfuscation of strings and cannot extract strings of routines that are not executed. If the string such as the attacker's IP address or the name of the malicious API is called in and the malicious APK is not deobfuscated, it may not be able to determine whether the APK is malicious.

2.3 Active Learning Based Ranking for Adversarial Android Malware

2.3.1 Active Learning

In the process of simulating human learning, machine learning uses existing knowledge to train a model to acquire new knowledge, and revise the model through the accumulated information to obtain a more accurate and useful new model. Different from passive learning, which passively accepts knowledge, active learning method can selectively acquire knowledge, which uses a certain algorithm to query the most useful unlabeled samples, which are later marked by experts, and then use the queried samples to train the classification model to improve the accuracy of the model.

Cohn's strategy for selecting samples in [69] is to directly calculate the classification error of the classifier on the test data set after adding the samples. This calculation has high complexity. In [70], Lewis used a classifier to select the samples with a class posterior probability close to 0.5 to join the training set. Each time he chose the most uncertain samples of the classifier to join the training set, the classification error of the classifier increased. In order to better evaluate the sample value, Seung [71] and Freund [72] proposed the committee voting selection algorithm. This method does not directly calculate the classification error but first establishes two or more classifications based on the existing class label data. Argamon-Engelson used the QBC method for natural language processing in [73] and achieved good results. Muslea et al. proposed the first multi-view active learning algorithm, Co-Testing, based on the Co-Training algorithm [74] in 2000, which became the basis of multi-view active learning.

2.3.2 Android Antivirus Products Penetration Testing

Penetration testing is the practice of testing computer systems, networks, or web applications to find security vulnerabilities that attackers might exploit. It can be automated or performed manually. Some work on Android-specific obfuscation tools has previously been proposed. For example, Rastogi *et al.* [40] presented Droid Chameleon, a tool for obfuscating Android apps. Zheng *et al.* [41] proposed ADAM, a framework for obfuscating Android apps and testing them on anti-malware products. Ghosh *et al.* [42] used a reflection call mechanism in Android application development to hide calls to sensitive APIs and access sensitive data. Nevertheless, this method can be identified by inspecting whether Java reflection calls are being used.

Harrison *et al.* [43] analyzes the code obfuscation technology that resists reverse engineering of Android applications. With this technology, obfuscation is divided into name obfuscation, control flow obfuscation, and instruction encoding obfuscation. Naming obfuscation converts the name of a package, class, or method into a meaningless string. However, since DEX bytecode is always stored in memory in the form of clear text during execution, an attacker can dynamically analyze the name obfuscation and implement the DEX bytecode loading logic [44].

Some other studies also focused on the effects of obfuscations on anti-malware products without proposing new obfuscation tools. Pomilia [45] studied the performance of nine anti-malware products on an obfuscated. Maiorca *et al.* [46] studied the effects of code obfuscated by a single tool on 13 anti-malware products. In order to find out the impact of code obfuscation strategies applied to Android applications, Mahmoud *et al.* [75] conducted an empirical study on the evaluation of antivirus products. The study used the Allatori [76], ProGuard [77], ADAM [41], DroidChameleon [40], and DashO [78] to obfuscate Android apps to generate adversarial samples. The authors first obfuscated 3,000 benign and 3,000 malicious apps to generate 73,000 obfuscated apps and then uploaded these samples to VirusTotal, a free online service that integrates over 60 antivirus scanners. However, the author did not explain what rules they chose based on the 73,000 obfuscated samples.

CHAPTER 3

Weakly-labeled Android Malware Family Clustering

3.1 Introduction

Android devices have represented around 80% of all smartphones since 2017 and are forecast to maintain their leadership with over 85% market share by 2020 [79]. The increasing popularity of Android devices has witnessed an unprecedented growth in emerging Android apps, which has also become the prime targets of attackers. It has been reported that the total number of Android malware had reached 856.52 million by the end of 2018. There were more than 137.5 million newly discovered malicious apps in 2018 (i.e., 350,000 new malware per day) [80]. Android malware is a major source of cyberattacks and is a serious threat to smartphone users.

Labeling a malicious app as an unknown or a variant of an existing family is important for identifying new threats, determining the severity of the threat, creating signatures for malware detection, malware triaging, and building reference datasets. Labeling malware is a non-trivial task. The raw labels reported by AntiVirus (AV) vendors are well-known to be inconsistent (e.g., two vendors may report two aliased family names for the same type of malware. `solimba` and `firseria` are aliases) without a standard naming convention (e.g., the conventions CARO [18] and CME

Table 3.1: Malware with empty labels

Clusters	AVCLASS		EUPHONY	
#App	# Empty-labeled	# percentage	#Empty-labeled	# percentage
9000	3066	34.06%	**1534**	17.04%

Table 3.2: Malware with controversial labels

#Gap = # vendor reporting the most frequent name - # vendor reporting the second most frequent name.

#Gap	0	1	2	3	4	≥ 5	# Apps	# Vendor
# App No.	**1790**	1389	825	594	438	2430	7466	67

[19] are not often used by AV vendors). Although manual inspection for malware labeling by an expert can provide an accurate solution, it is extremely costly in practice due to the huge number of apps being released every day.

Existing efforts. To address these issues, the recently proposed clustering approaches, such as AVCLASS [1] and EUPHONY [2], perform an automatic malware labeling based on the outputs from a collection of AntiVirus vendors in VIRUSTOTAL [7]. Due to the inconsistency among the raw labels reported by a wide variety of AV vendors, the existing approaches normally perform a pre-processing to extract the family names from the raw labels based on vendor-specific or self-defined heuristic rules [1, 2], including generic token removal and alias token reduction. Plurality voting [81] is then applied to disambiguate the inconsistent family names.

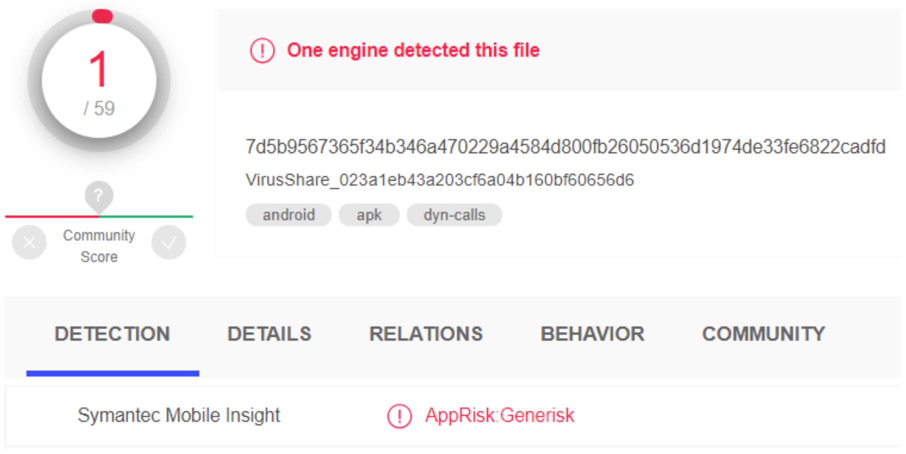
Limitations and our observations. These *raw-label-based* approaches rely heavily on the original labels from the AV vendors. A substantial number of *weakly-labeled malware* are unable to be clustered because their raw labels are often incomplete, inconsistent and overly generic, as reported by incompatible commercial vendors with different capabilities in the presence of rapidly evolving malware.

We define two types of weakly-labeled malware based on the results from the state-of-the-art approaches [1, 2] with their statistics given in Tables 3.1 and 3.2: (1) *Malware with empty label.* The tokens from the raw labels being recognized as generic tokens (e.g., `apprisk`) or unable to be parsed by the rules of AVCLASS and EUPHONY are discarded, resulting in weakly-labeled apps without a family name. Figure 3.1 illustrates an concrete example of malware with empty label. In addition, for example, as listed in Table 3.1, AVClass and Euphony are unable to produce a family name for 34.06% and 17.04% of the total 9000 apps which were recently uploaded to VirusTotal between Mar. 2017 and Feb. 2018, resulting in **1534** weakly-labeled malware. (2) *Malware with controversial labels.* The plurality voting strategy is also hard to disambiguate inconsistency between two family names reported by a close number of vendors among the 7466 apps which have family names extracted from the raw labels by Euphony. In Figure 3.2, an example of malware with controversial labels is given. Table 3.2 shows that the top two most frequent names are reported by an equal number of vendors for **1790** apps (24%), causing the plurality voting to be less confident for these weakly-labeled malware compared to their strongly-labeled counterpart with an uncontroversial family name reported by the majority (e.g., 80%) of all available vendors in VirusTotal [7].

An example of malware with empty label is given in Fig 3.1. Only one AV vendor (`Symantec Mobile Insight`) reports it as malware. The raw label `AppRisk.Generisk` given by this vendor is very generic. According to the rules of AVClass and Euphony, the raw label `AppRisk.Generisk` is parsed as two generic tokens `AppRisk` and `Generisk` with no actual family information, thus returning an empty label after their generic token removal phases.

An example of malware with controversial labels is given in Fig 3.2. There are seven AV engines report this app as malware. According to Euphony, five of them are generic labels (it becomes empty labels after generic token removal by Euphony). The remaining two AV engines label this malware as two different families, i.e., `dinehu` by `NANO-Antivirus` and `Emagsoftware` by `Sophos AV`. Thus, the family name of this app becomes controversial if the malware labeling is purely based on the raw labels produced by AV vendors.

Inspired by the recent advances in DRL (Deep Representation Learning), this chapter proposes Andre, a new hybrid representation learning approach to clustering weakly-labeled malware by utilizing multiple sources of data (including raw labels from AV vendors, meta-info of apps and results from source code analysis). Andre jointly learns a hybrid representation that allows heterogeneous information to be integrated into one neural network pipeline that distills the discriminative fea-


One engine detected this file

7d5b9567365f34b346a470229a4584d800fb26050536d1974de33fe6822cadfd

VirusShare_023a1eb43a203cf6a04b160bf60656d6

Symantec Mobile Insight | AppRisk:Generisk

Figure 3.1: An example of malware with empty label.

tures for accurate clustering. ANDRE is based on a new Android malware network, in which every node represents an app whose label contains its meta-info and the raw reports from AV vendors, and every edge denotes the similarity between two apps inferred by our static analysis which exploits a pairwise analysis of code similarity. Weakly-labeled malware in the network, together with existing strongly-labeled malware, are fed into our hybrid representation learning model to embed all the nodes on the network into a continuous and low-dimensional space that preserves comprehensive heterogeneous information.

The learned representation is then used for our outlier-aware clustering to partition the weakly-labeled malware into known and unknown families. The malware whose malicious behaviours are close to the existing families on the network, are further classified using a three-layer Deep Neural Network (DNN). The unknown malware are clustered using a standard density-based clustering algorithm.

Our solution. The model leverage multisource info and map them into common continuous and low dimensional space. To learn the optimal representation, the tri-party information is jointly given to the neural network and then enhance each other mutually.

Deep representation learning is a special machine learning approach that extract a high level of features from low-level data. The input is raw features(meta info, label info, code similarity,etc). The representation learns higher the abstraction level of the features(higher-level features). Through mapping high-dimensional data into a lower-dimensional space, representation learning can learn from sparse input data.

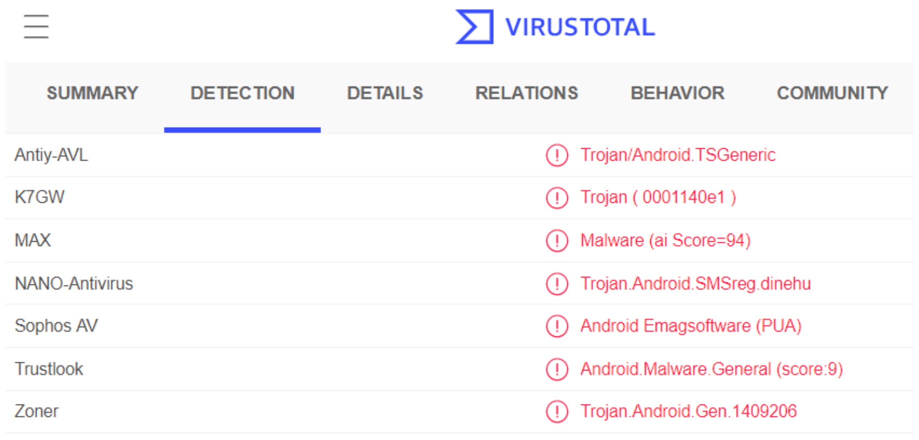
In this chapter, we attempt to use the representation learning method to cluster malware and prove its advantage in corresponding scenario. Based on these insights, we propose ANDRE, a tri-parity deep network representation model which learn from three parties' information: Android malware network structure, malware meta info content, and malware label info(if available) to jointly learn optimal malware representation. The model leverage multisource info and map them into common continuous and low dimensional space. Then the deep neural network model will learn the tri-party information, in the meanwhile, obtain the optimal representation with enhancing each other mutually.

The rest of the chapter is structured as follows. Section 3.2 defines the malware clustering problem and introduces the overall framework of ANDRE. Section 3.3 details our approach including feature extraction, network construction, representation learning and outlier-aware clustering.

Problem Definition. An Android malware network is represented as $G = (N, E, W, C)$, where the node set N = $\{n_1, n_2, \ldots, n_{|N|}\}$ denotes a set of Android malware (including both strongly- and weakly-labeled apps), and an edge $e_{i,j} = (n_i , n_j) \in E$ between two nodes encodes the code similarity between two apps. Each node n_i is associated with content information d_i consisting of a sequence of word tokens $w_j \in d_i$ extracted from the app's meta-info and the raw labels produced by AV vendors. We use $W = \{w_1, \ldots, w_{|W|}\}$ to denote the words of all nodes in this network.

Every node n_i has a label $l_i \in C = L \cup U$, where U denotes a set of labels with no family name for prediction purposes, and L are known family names of the strongly-labeled malware (either from ground-truths in Drebin or downloaded from VIRUSSHARE with each app's unique family name reported by over 80% of all available vendors). The numbers of ground-truth and weakly-labeled apps are configurable in the network. If the label set $L = \emptyset$, i.e., $C = U$, the representation learning becomes purely unsupervised, and our proposed solution is still valid but imprecise.

Our hybrid representation learning problem is formulated as maximizing an



Antiy-AVL	Trojan/Android.TSGeneric
K7GW	Trojan (0001140e1)
MAX	Malware (ai Score=94)
NANO-Antivirus	Trojan.Android.SMSreg.dinehu
Sophos AV	Android Emagsoftware (PUA)
Trustlook	Android.Malware.General (score:9)
Zoner	Trojan.Android.Gen.1409206

Figure 3.2: An example of malware with controversial labels.

objective function $\mathcal{J}$ (Equation 3.1), which aims to jointly learn a k-dimensional vector $\mathbb{V}_{n_i} \in \mathbb{R}^k$ (k is a smaller number) for each node n_i in the network, such that nodes, which are neighbors based on the network topology or have similar meta-info or family names are close to one another in the latent embedding space.

$$\begin{aligned}\mathcal{J} = &\alpha_1 \sum_{i=1}^{|N|} \sum_{j=1}^{|N|} \mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(\mathbb{V}_{n_i}, \mathbb{V}_{n_j}) + \alpha_2 \sum_{i=1}^{|N|} \sum_{p=1}^{|W|} \mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(\mathbb{V}_{n_i}, \mathbb{V}_{w_p}) \\ &+ \alpha_3 \sum_{q=1}^{|L|} \sum_{p=1}^{|W|} \mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(\mathbb{V}_{w_p}, \mathbb{V}_{c_q})\end{aligned} \tag{3.1}$$

where $\mathbb{V}_{n_i}$, $\mathbb{V}_{w_p}$ and $\mathbb{V}_{c_q}$ denote the learned representation vectors of the three entities, i.e., node n_i, word w_p and family c_q in the representation space $\mathbb{R}^k$. $\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}$, $\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}$ and $\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}$ are affinity functions to capture the correlation between two entities in $\mathbb{R}^k$. α_1, α_2 and α_3 are the weights that balance network structure, meta-info, and family information ($\alpha_1 + \alpha_2 + \alpha_3 = 1$).

3.2 Framework Overview

Figure 3.3 gives the overview of our framework, which consists of the following three major components, where n_i represents a single malicious app, and c_i denotes a known family name if n_i is a strongly-labeled. c_i is empty if n_i is weakly-labeled for prediction purposes. w_j denotes a word in the meta-info associated with a node.

3.2.1 Feature Extraction

There are two steps for extracting features. (1) Code similarity analysis. ANDRE implements a pairwise comparison scheme to dissect the similarities between apps. Two apps are provided as inputs and our method yields a similarity profile for each pair. The similarity profile summarizes similarity facts relating to similarity scores at file level, which means calculating similarity score for two source code files. The method builds an invertd-index to quickly calculate the similarity score of a pair of Android source code files. In addition, there is also a filtering heuristics to reduce the size of the index, which reduces the number of pairs needed to evaluate the similarity scores. A large portion of an Android app contains standard Android and safe third party libraries [33], which can be seen as noise in our representation learning. Following [33], we maintain a whitelist of common libraries to remove those library-related code segments from the application code of a malicious app in our code similarity analysis. Whitelist is a standard way for reducing noise in code similarity analysis. In addition, whitelist also provides a flexible and customized way for adding any new APIs which are safe.

(2) Analyzing the Android manifest files. We extract meta-info of each malware app from its manifest files, including package names, API versions, launcher activities, permissions, etc, which are associated with the corresponding node in the network. In addition, the raw labels from the existing AV vendors are also attached to each node in our network. Finally, all the ground-truths (strongly-labeled) malware are labeled with their unique family names.

3.2.2 Hybrid Representation Learning

An Android network G is constructed from the code similarity analysis, where each edge encodes the similarity between node n_i and n_j. The meta-info and raw labels associated with node n_i are represented by d_i. Our network G contains both weakly- and strongly-labeled (as the ground-truths) malware nodes with configurable portions of both kinds. The Hybrid (or multimodal) Representation Learning (HRL) module [10, 82, 83] jointly embeds nodes N, words W from the meta-info, and family names L in a low-dimensional space, so that the affinity of heterogeneous information can be captured. The malware clustering can be accurately performed based on the comprehensive representation space via Equation 1, through which the following three relationships are preserved.

(1) *Inter-app correlation* $\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(\mathbb{V}_{n_i}, \mathbb{V}_{n_j})$, which captures the relationship between two apps via a neural network based on the randomly generated sequences (random walks) from the network structure [35], where each sequence $s = n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_n$ can be seen as a phrase in natural language model. Given a node n_j in each random walk sequence s within a sliding window b, the neural network maximizes the log-likelihood of observing a set of neighboring nodes $\{n_{j-b}, n_{j-b+1}, \cdots, n_{j+b-1}, n_{j+b}\}$, so that the representation vectors $\mathbb{V}_{n_i}$ and $\mathbb{V}_{n_j}$ are close to each other if $e_{i,j} \in E$, i.e., $\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(\mathbb{V}_{n_i}, \mathbb{V}_{n_j})$ is maximized.

(2) *Node-word correlation*, which maximizes the co-occurrence of a node and a word in the meta-info, i.e., maximizing the affinity value of $\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(\mathbb{V}_{n_i}, \mathbb{V}_{w_p})$ in the embedding space, modeling the fact that if a word w_p from the meta-info appears in node n_i, then the two vectors $\mathbb{V}_{n_i}$ and $\mathbb{V}_{w_p}$ are near in the representation space.

(3) *Label-word correlation*, which maximizes the co-occurrence of a family name and a word in the meta-info, i.e., maximizing $\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(\mathbb{V}_{w_p}, \mathbb{V}_{c_q})$, such that the label c_q and word w_p are forced to be closed to each other in the resulting representation space.

Lastly, the above three relations are jointly learned in a unified mode in an iterative manner. We further employ a hierarchical softmax modeling [84] to reduce the time complexity of our model, enabling ANDRE to scale to large malware datasets.

3.2.3 Outlier-aware Weakly-labeled Malware Clustering

Given our learned hybrid representation space $\mathbb{R}^k$, this step performs prediction of a weakly-labeled app in the network G. When predicting its family name, this malware may fall into an existing family in L from the strongly-label malware in G or it may be an unknown type of malware with different behaviors which are different from any known families in L since the unknown malware types do exist and the ground-truth labels from strongly-labeled malware in the network may be limited. To address this issue, ANDRE performs an outlier-aware clustering that first employs outlier detection to partition all the weakly-labeled malware apps into (1) inlier apps whose behaviors are close to those of known families, and (2) anomalous apps that unlikely belong to any existing family in the network. Malware apps in the outlier set are clustered using a density-based algorithm, while apps in the inlier set are fed into a designed neural network to train the multi-classifier for classifying these malware into known families. The neural network consists of three layers, where the first two 1024-node layers comprise a dropout followed by a dense layer with a parametric rectified linear unit (ReLU) activation function in the first two layers and the sigmoid function in the third layer, which is also used for prediction.

3.3 Our Approach

This section introduces the detailed approaches of ANDRE, including Android malware feature extraction, hybrid representation learning and outlier-aware clustering for weakly-labeled malware.

3.3.1 Android Malware Feature Extraction

The aim of our feature extraction is to build the network via code similarity analysis between two nodes (apps) and associate each node with its meta-info extracted from its corresponding app.

Our source code analysis and meta-data extraction are complementary. Performing code similarity analysis on possible malicious code segments from two programs is helpful for grouping malware apps with similar runtime behaviors. The meta-info of an Android app, including permissions, Android services, activities, providers, receivers, intent filters, security settings and referenced libraries are important building blocks for understanding about the special characteristics of an app. Combining code similarity and meta-info for constructing the network provides more comprehensive understanding of app relationships.

Code similarity analysis. Directly applying the existing code clone analysis, such as SourcerCC [85] and Deckard [86] on Android apps to obtain their similarity information is ineffective, because an Android app commonly contains a large portion of safe library code, either by invoking Android SDK APIs or third party libraries. The malicious code segments which reflect the malware type may be hidden in the application code. The safe (library) code segments may become noisy, resulting in inaccurate identification of code similarity between the two malware families. In addition, some malware apps are developed by repackaging an existing benign app [87]. By injecting two different types of malicious code into one benign app, the two repackaged apps can be of different malware families. Therefore, the benign code segments from Android SDK and third-library are the major noises, impeding analyzing code similarity to differentiate the two malware families.

Our code similarity analysis consists of four steps. First, all the apks are decompiled using `dex2jar` by converting their original DEX files to Java files for our source code analysis. Next, we perform a common library removal to reduce as many noise as possible, and our token-based code similarity analysis is performed to group apps based on file-level similarity. We then perform fine-grained code-block-level similarity analysis by considering existing available malicious payloads [33, 88] to refine the similarity scores between two apps. Lastly, an edge in the network is connected between two apps if their similarity score is above a pre-defined threshold.

Common library removal. Previous study [89] shows that over 60% of Android application code (in terms of low-level bytecode instructions after compilation) is contributed by library code. To address the noises introduced by a large portion of library code when conducting code similarity analysis, we follow [23, 33, 90, 91] to perform a common library removal by maintaining comprehensive whitelists of common libraries [92] to exclude library-related code segments prior to our similarity analysis.

Code similarity. Our code similarity analysis applies a bag-of-words-based code clone approach [85], which has been shown to be the most efficient strategy and has a good precision for scaling to large code bases. Tokenization is first performed to remove comments, white space, and terminal. A token is extracted from each source file into Java keywords, literals, and identifiers. A string literal is split on whitespace and operators are not included. Tokens in known malicious payload code segments [33, 88] are given higher weights for code similarity analysis. The source code of an Android app n is represented as a set of code blocks (basic blocks of the control-flow graph of a program) $Source(n) = \{B_1, ..., B_{num}\}$ with each block B_i denoting a bag-of-tokens $B_i = \{T_1..., T_k\}$. One token may appear multiple times in a block, therefore each token is qualified with its occurrence frequency inside a block, $T_j = (token, frequency)$ to differentiate between the frequency of words in the bag-of-words model. Formally, given two apk files A_x and A_y, a similarity function f, and a threshold θ, the aim is to find all the code block pairs A_x.B and A_y.B s.t $f(A_x.B, A_y.B) \geq [\theta \cdot max(|A_x.B|, |A_y.B|)]$.

There are several choices of similarity function for measuring the similarity between two code pieces, we use the Jaccard index, or Jaccard similarity, which is defined as the size of intersection divided by the size of the union of two sets. The similarity of two code blocks B_x and B_y is defined as follows:

$$J(B_x, B_y) = \frac{|B_x \cap B_y|}{|B_x \cup B_y|} = \frac{|B_x \cap B_y|}{|B_x| + |B_y| - |B_x \cap B_y|} \tag{3.2}$$

Meta-Info extraction. Our meta-info is mainly extracted from `AndroidManifest.xml`, which is the key file at the root of an Android project. It provides all the essential information of an app to the building tools, Android OS, and app stores. Once an app is launched, `AndroidManifest.xml` is the first file to be consulted by the Android system, providing the first-hand information to understand the characteristics and security settings of the app.

The detailed content of a manifest file is illustrated in Table 3.3, including (1) `permissions`, which are responsible for protecting the application from accessing any protected parts, (2) `instrumentation` classes, which provide profiling and other dynamic monitoring information, (3) `application-level Android APIs` that an app is going to use, (4) four types of Android components: `activity`, `service`, `content provider`, and `broadcast receiver`. The names of an app's components may help identify the known malware components. (5) `hardware component`, which is helpful to identify malicious behaviors reflected by access requests to specific device components, e.g., touchscreen, camera, or sensors. (6) `intent and intent filter`, which can be used to trigger malicious activities, thus it is also necessary to be collected, and (7) `package name, version, referenced libraries`.

3.3.2 Android Network Representation Learning (ANDRE)

This section presents our malware representation learning which jointly exploits the network structure, the meta-info, and vendors' labels to embed heterogeneous information into a latent feature space for each node in the network.

The process of ANDRE consists of two steps:

- Network node (app) sequence generation with random walks. This step randomly generates a set of walks on the Android malware network, where each sequence starts with a node n_i and randomly jumps to other nodes at each iteration. The random walk sequences capture the topological relationships between nodes.

- Neural network architectures. This step encodes each node (app) into a vector representation by preserving multi-source information from (1) an android sequence which captures the inter-app correlation, (2) the meta-info which represents the node-word correlation, and (3) the label information which records the label-word correlation. Lastly, a hybrid representation model is built by jointly learning the above three correlations.

The proposed joint neural network model is illustrated on the right of Figure 3.4. The DEEPWALK approach learns the network representation based only on the network structure. Our hybrid method couples two neural networks to learn the representation from three parties (i.e., node structure, meta-info, and label information from strongly-labeled malware) to capture the inter-node, node-word, and label-word relationships. The input, projection, and output indicate the input layer, hidden layer, and output layer of a neural network model.

It has key properties:

Inter-node correlation. Assuming that neighbor nodes are highly correlated and statistically dependent on each other, the upper layer of ANDRE exploits the network structure from a set of generated random walk sequences.

Node-word correlations. The lower layer of ANDRE captures the relations between nodes $n_i \in N$ and the meta-info, which is considered as a document with a set of words $w_j \in W$. If a word is used to describe a node, the node and the word are correlated.

Label-word correlation. We use the labels $c_i \in C$ from strongly-labeled malware (a ground-truth) and its corresponding node $n_i \in N$ as input and learn the label vector and word vector simultaneously, so that the label-word correlation can be well captured to improve the learning model.

Joint training model. We integrate these two layers by the node n_i in the model. The three kinds of correlations are jointly learned in a unified way, so that they can benefit each other and ultimately converge to a steady stage.

3.3.3 Model Details

Inter-node correlation. To capture the inter-node correlation, we assume that the nodes with linkages (edges) in a network are statistically dependent on each other. This idea is inspired by DEEPWALK [35], which extends the SKIP-GRAM [34] algorithm.

The SKIP-GRAM model [34] is a language model that learns the word embedding by exploiting word orders in a sequence and assuming that words close to one another are statistically more related. Due to its simplicity and efficiency, The SKIP-GRAM model is widely used in many NLP tasks. Given a word w_m, Skip-Gram maximizes the log-likelihood of w_m's surrounding words within a certain window b.

$$\mathcal{J}_1 = \sum_{m=1}^{T} \log \mathbb{P}(w_{m-b} : w_{m+b} | w_m) \tag{3.3}$$

where b is the window size and $w_{m-b} : w_{m+b}$ is a word sequence in a window, which excludes the word w_m itself. The probability $\mathbb{P}(w_{m-b} : w_{m+b} | w_m)$ is defined as:

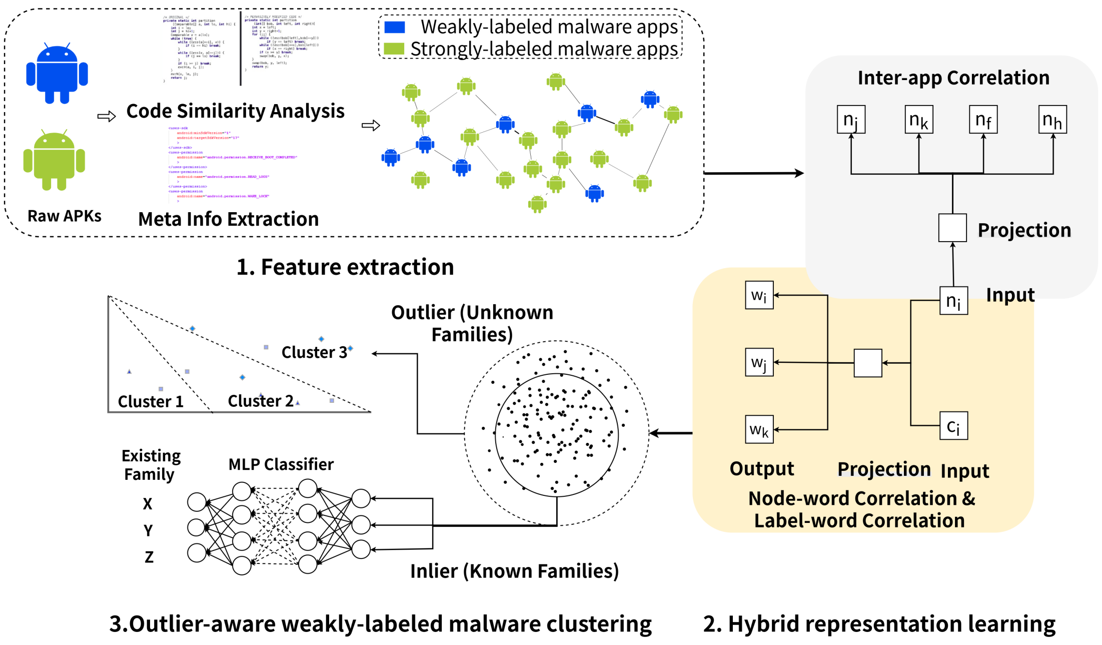


Figure 3.3: The overview of ANDRE.

$$\prod_{-b\le j\le b, j\ne 0} \mathbb{P}(w_{m+j}|w_m) \tag{3.4}$$

Equation (3.4) makes the assumption that the words in a window $w_{m-b} : w_{m+b}$ are independent of each other. $\mathbb{P}(w_{m+j}|w_m)$ is defined as:

$$\mathbb{P}(w_{m+j}|w_m) = \frac{exp(\mathbb{V}_{w_m}^\top \mathbb{V}'_{w_{m+j}})}{\sum_{w=1}^{W} exp(\mathbb{V}_{w_m}^\top \mathbb{V}'_{w})} \tag{3.5}$$

where $\mathbb{V}_w$ and $\mathbb{V}'_w$ are the *input vector* and *output vector* of w. Once the training is finished, the *input vector* $\mathbb{V}_w$ is used as the final representation of word w.

DEEPWALK [35] extends SKIP-GRAM and employs the neural language model to learn the embedding for a network. Specifically, DEEPWALK generates a set of random walk sequences S based on the network structure. Each random walk $s = n_{i-b} \rightarrow \ldots n_i \ldots \rightarrow n_{i+b}$ is regarded as a sentence with a window of size b in the natural language model, and each node n_i is considered as a word. Then DEEPWALK employs the SKIP-GRAM algorithm [34] on the node sequences to obtain a latent representation for each node. This is achieved by solving an objective function which maximizes the log-likelihood of the observed nodes on a target node n_i for all random sequences $s \in S$.

$$\begin{aligned}\mathcal{J}_2 &= \sum_{i=1}^{|N|}\sum_{s\in S}\log\mathbb{P}(n_{i-b}:n_{i+b}|n_i)\\ &= \sum_{i=1}^{N}\sum_{s\in S}\sum_{-b\leq j\leq b, j\neq 0}\log\mathbb{P}(n_{i+j}|n_i)\end{aligned} \tag{3.6}$$

The DeepWalk neural network is illustrated on the left of Figure 3.4. It exploits the network structure for learning representations without considering other information (such as meta-info and label-info), which is exploited by our model.

Node-word correlation. To enable joint learning together with inter-node correlations, Andre captures the node-word correlations as depicted in the right lower panel in Figure 3.4, which applies a Doc2Vec style model built on top of Skip-Gram and CBOW (Continuous Bag-Of-Words) [34], with the aim of learning the distributed representation for a document. In our setting, for a node n_i, we learn the input node vector $\mathbb{V}_{n_i}$ and output word vector $\mathbb{V}'_{w_j}$ based on meta-info $d_i = [w_0, w_1, \ldots, w_{|d_i|}]$ associated with n_i using a sliding window of size b for repeatedly picking a sequence of words centering w_j within d_i.

Label-word correlation. We enable semi-supervising by leveraging uncontroversial family names from strongly-labeled malware, benefiting from the knowledge of existing AV vendors. By applying Doc2Vec, we use the ground-truth label information together with the meta-info as inputs and simultaneously learn the input label vector $\mathbb{V}_{c_i}$ of node n_i and output word vector $\mathbb{V}'_{w_j}$ based on the meta-info associated with n_i, modeling the correlation between the nodes' labels and the nodes' meta-info.

Since the first correlation is captured via DeepWalk and the last two correlations are modeled via Doc2Vec, it can intuitively be seen that our hybrid learning couples these two modelings using two panels, as illustrated in Figure 3.4. The node n_i shared by both panels indicates that n_i is influenced by the two models to produce a hybrid representation which preserves the heterogeneous information and the relations between the three parties (e.g., node sequences from random walks, word sequences from meta-info, and label information).

Joint learning model. Given a network G consisting of nodes $\mathrm{N} = \{n_1, n_2, \ldots, n_{|N|}$

Table 3.3: Meta-data information extracted from an Android app

Type	Description
Permission	Permissions constitute an important security feature in Android apps. A user has to grant them to install applications or access particularly sensitive data.
Instrumentation	Declares the code used to test this package or other package directive components. A manifest can contain zero or more of this element.
Application	Contains the root node of the application-level component declaration in the package and contains global and default properties in the application, such as tags, icons, themes, necessary permissions, and more.
Activity	Activity is the primary Android component used to interact with users.
Service	A service is a component that can run any time in the background.
Content provider	A content provider is a component that is used to manage persistent data and publish it to other applications.
Broadcast receiver	The receiver enables the application to obtain data changes or operations that occur even if it is not running.
Intent/intent-filter	The IntentFilter is formed by declaring the Intent values supported by the specified set of components.
Action	The intent action supported by the component.
Category	Intent Category supported by the component.
Type	Intent data MIME type supported by the component.
Schema	Intent data URI scheme supported by the component.
Authority	Intent data URI authority supported by the component.
Path	Intent data URI path supported by the component.

}, our hybrid learning model in Equation 3.7 implements the pre-defined objective function in Equation 3.1 by considering the three aforementioned correlations. The aim is to jointly learn the following three affinity functions with random walks S generated for node n_i, and a sliding window for a sequence of nodes or words.

$$\begin{aligned}\mathcal{J} = (1-\alpha)\sum_{i=1}^{|N|}\sum_{s\in S}\sum_{-b\leq j\leq b, j\neq 0}\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(n_{i+j}, n_i)\\ +\,\alpha\sum_{i=1}^{|N|}\sum_{-b\leq j\leq b}\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(w_j, n_i)+\alpha\sum_{i=1}^{|L|}\sum_{-b\leq j\leq b}\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(w_j, c_i)\end{aligned} \tag{3.7}$$

where α is the weight that balances the network structure, meta-info, and label information. b is the window size of a node or a word sequence, and w_j indicates the j-th word in a contextual window. Given Equation 3.7, the first term computes the affinity function $\mathcal{A}(n_{i+j}, n_i)$, and the log-likelihood probability of observing surrounding nodes given node n_i, using the log-likelihood softmax functions as Equation 3.8:

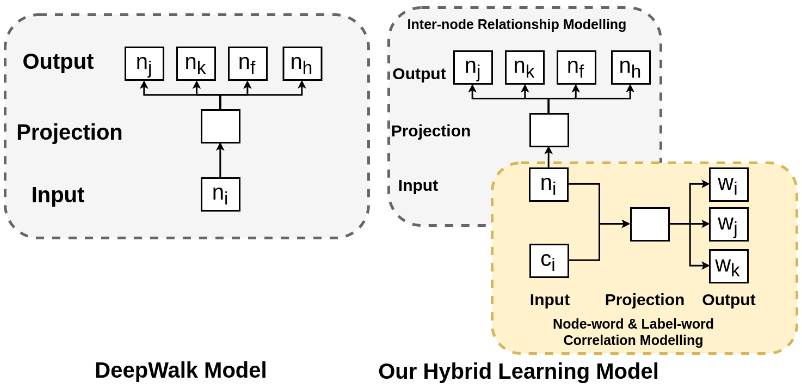


Figure 3.4: The DeepWalk (Skip-Gram) method vs our proposed hybrid learning method.

$$\mathcal{A}^{\mathcal{N}\circ\mathcal{N}}(n_{i+j}, n_i) = \log \mathbb{P}(n_{i+j}|n_i) = \log \frac{exp(\mathbb{V}_{n_i}^{\top}\mathbb{V}'_{n_{i+j}})}{\sum_{x=1}^{|N|} exp(\mathbb{V}_{n_i}^{\top}\mathbb{V}'_{n_x})} \tag{3.8}$$

where $\mathbb{V}_{n_x}$ and $\mathbb{V}'_{n_x}$ are the input and output vector representations of node n_x. $|N|$ is the number of the nodes in the network. The probability of observing contextual words $w_{i-b} : w_{i+b}$ given current node n_i is:

$$\mathcal{A}^{\mathcal{N}\circ\mathcal{W}}(w_j, n_i) = \log \mathbb{P}(w_j|n_i) = \log \frac{exp(\mathbb{V}_{n_i}^{\top}\mathbb{V}'_{w_j})}{\sum_{x=1}^{|W|} exp(\mathbb{V}_{n_i}^{\top}\mathbb{V}'_{w_x})} \tag{3.9}$$

where $\mathbb{V}'_{w_j}$ is the output representation of word w_j, and $|W|$ is the number of distinct words on the whole network. Similarly, the probability of observing the words given a class label c_i is then defined as:

$$\mathcal{A}^{\mathcal{L}\circ\mathcal{W}}(w_j, c_i) = \log \mathbb{P}(w_j|c_i) = \log \frac{exp(\mathbb{V}_{c_i}^{\top}\mathbb{V}'_{w_j})}{\sum_{x=1}^{|W|} exp(\mathbb{V}_{c_i}^{\top}\mathbb{V}'_{w_x})} \tag{3.10}$$

Equations 3.9 and 3.10 reflect the correlation between nodes and meta-info, and the correlation between meta-info and label information, so that Andre jointly learns the output representation vector $\mathbb{V}'_{w_j}$ of word w_j, which will propagate back to influence the input representation of $n_i \in N$ in the network as also illustrated in Figure 3.4. As a result, the node representation (i.e., the input vectors of nodes) is enhanced by both the meta-info and the label information.

Training hybrid learning model. We train our model Equation 3.7 using stochastic gradient ascent, which is a standard solution for training. However, computing the gradients in Equations 3.8, 3.9 and 3.10 is expensive, as the computation is proportional to the number of nodes and words in the network G. To address this issue, we resort to hierarchical softmax [10, 84], which reduces the time complexity to $O(|W|log(W) + Nlog(N))$.

The hierarchical model in our algorithm uses two binary trees, one with distinct nodes as leaves, and another with distinct words and labels as leaves. The trees are built using Huffman algorithm, so that each vertex in a tree has a binary code, in which more frequent nodes (or words) have shorter codes. There is a unique path from the root to each leaf in a tree. The interval vertices of the trees are represented as real-valued vectors with the same dimension as the leaves. Instead of enumerating all nodes in Equation 3.7 in each gradient step, we only need to evaluate the path from the root to the corresponding leaf in the Huffman tree. Suppose the path to the leaf node n_i is a sequence of vertices $(l_0, l_1, ..., l_h)$ reaching n_i, where l_h is n_i and l_0 is the root node in the Huffman tree. The probability is computed as follows:

$$\mathbb{P}(n_{i+j}|n_i) = \prod_{t=1}^{h} \mathbb{P}(l_t|n_i) \tag{3.11}$$

$$\mathbb{P}(l_t|n_i) = \sigma(\mathbb{V}_{n_i}^{\top} \mathbb{V}'_{l_t}) \tag{3.12}$$

where $\mathrm{P}(l_t|n_i)$ is a binary classifier with $\sigma(.)$ as the sigmoid function, and $\mathbb{V}'_{l_t}$ is the representation of node l_t, which is the parent of n_i in the Huffman tree. Thus, the time complexity is reduced to $O(NlogN)$. Likewise, we can use hierarchical softmax technique [13] to compute the words and labels in Equations 3.9 and 3.10.

3.3.4 Outlier-aware Android Malware Clustering

Outlier detection (a.k.a. anomaly detection [93]) detects rare samples that do not meet the expected patterns or behaviors of the majority of samples in the dataset. The technique is essentially a density-based outlier detection algorithm that constructs a graph of the data using nearest neighbors, instead of calculating local densities. A malware sample whose malicious behaviours are closed to those of known families in the network G will be classified into an existing family in L, otherwise, it will be identified as an unknown type of malware. This is due to the fact that the unknown malware types do exist and the number of ground-truth labels from strongly-labeled malware in the network may always be limited.

Finally, the outlier set of the collected malware apps is clustered to produce unknown family clusters, while the inlier set is fed into the designed Multi-layer Perceptron (MLP) classifier (Supervised Neural Network) to train the multi-classification model, thereby classifying these malware into known families.

3.4 Evaluation

The objective of our evaluation is to show that ANDRE is effective in clustering weakly-labeled malware that cannot be clustered by AVCLASS and EUPHONY, while achieving comparable accuracy with the state-of-the-art tools evaluated using the ground-truth samples in the Drebin dataset.

3.4.1 Experimental Setup and Implementation

To evaluate the effectiveness of ANDRE, 5,416 ground-truth malware samples from the Drebin dataset [22] and 9,000 recent malware from VIRUSSHARE [94] (uploaded between March 2017 and October 2018) were used. The Android apks were first decompiled to Java files using `dex2jar` for our source code analysis. Note that there are 5560 malware samples in the Drebin dataset, but only 5416 samples were used since the remaining 144 samples could not be correctly decompiled by `dex2jar`. Out of the 9,000 recently collected malware from VIRUSSHARE, 4314 are smaller than 5MB, 1969 are between 5MB and 10MB, and 2717 are greater than 10MB.

To build the edge relations between nodes (apps) of the network, we adopted the bag-of-words model [85] to perform code similarity analysis for all pairs of malware apps after excluding their library-related code, following [33] by using a self-maintained whitelist containing common third-party libraries [92] and the available malicious payload [33, 88]. An edge is connected between two nodes if their similarity score is above 0.8 (with the maximum score 1 using Jaccard similarity in Equation 3.2 after normalization).

For each app in the network, we use the ANDROID ASSET PACKAGING TOOL [95] to extract meta-info of an app. The raw labels reported by Anti-Virus vendors were obtained by uploading an app or its hash value to VIRUSTOTAL [7], an online malware scanning service which integrates a set of existing commercial Antivirus vendors, such as `Comodo` and `Kaspersky`. Given VIRUSTOTAL's reports, we can identify weakly-labeled malware, including malware with no label (1534) and malware with controversial family names (1790) by using EUPHONY. The results are shown in Table 3.1 and Table 3.2. Table 3.1 gives the number of malicious apps which do not have a family name after clustering by EUPHONY and AVCLASS due to overly generic raw labels being reported by AV vendors. The 9000 malicious apps are from VIRUSSHARE uploaded between 03/2017 and 02/2018. Table 3.2 gives the number of malicious apps whose top two frequent families reported by a close number of vendors based on plurality voting using EUPHONY. Of the 7466 apps which have family names (excluding the 1534 empty labeled apps in Table 3.1), plurality voting is not confident in 1790 apps since the two most frequent names are reported by an equal number of vendors for each app.

We implemented DEEPWALK, DOC2VEC and our hybrid learning models to embed the constructed network using Python. Our outlier detection approach was implemented based on TOPOLOGICAL ANOMALY DETECTION (TAD) [96]. To classify the inlier known families, we applied and compared different classification algorithms, including the neural network classifier MULTI-LAYER PERCEPTRON (MLP) [97] and traditional classification algorithms SVM [98], KNN [99], DSTREE [100] and RDFOREST [101] whose implementations are available from Scikit-learn library [102].

3.4.2 Evaluation Methodology

We evaluate the effectiveness of our approach from the following four aspects: (1) comparing ANDRE with the state-of-the-art tools using Drebin's ground-truth samples (a typical dataset widely used by existing clustering tools) to show the effectiveness and applicability of our representation learning approach for clustering all types of malware (not limited to weakly-labeled malware) with good precision, (2) comparing ANDRE against different baseline settings (i.e., different representation learning models including DEEPWALK and DOC2VEC) to show whether our hybrid learning model can obtain a promising level of accuracy by preserving heterogeneous information, (3) comparing different classification models given the learned representation, and (4) validating our outlier-aware malware clustering results with comprehensive case studies to show ANDRE's effectiveness in clustering weakly-labeled malware.

Table 3.4: Comparison with AVCLASS [1] and EUPHONY [2]. The accuracy, F1 and recall data are directly from their papers

Method	Accuracy	F1 Score	Recall
AVCLASS	95.2%	93.9%	92.5%
EUPHONY	95.0%	95.5%	96.1%
ANDRE	93.1%	93.3%	93.1%

3.4.3 Comparing with State-of-the-art Tools and Different Baseline Settings

Comparing with non-learning approaches. We compare ANDRE with the two state-of-the-art tools AVCLASS and EUPHONY by using ground-truth samples from Drebin to show ANDRE's applicability and effectiveness in clustering malware which are not weakly-labeled. Our hybrid representation learning approach randomly selects 70% of malware as the training set and 30% of samples in the dataset for testing. Table 3.4 shows that ANDRE achieves 93.1% for accuracy, 93.3% for F1 score and 93.1% for recall, which are comparable or slightly less than the results of the two tools (reported in their papers [1] and [2]).

Note that we do not claim that our learning-based approach is superior to the non-machine-learning methods in clustering malware apps that are not weakly-

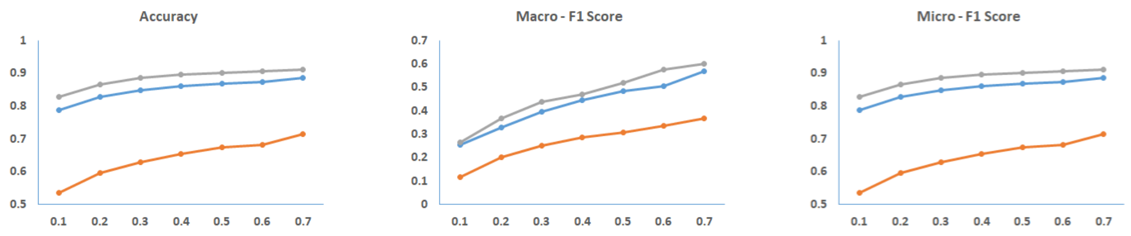


Figure 3.5: Performance comparison with respect to different learning models.

labeled. This is because plurality voting does work well for strongly-labeled samples with uncontroversial raw labels from AV vendors after label preprocessing, such as generic token removal and alias detection [1]. Rather, our aim is to show that our approach successfully preserves necessary information for effective clustering and can achieve comparable accuracy with the existing tools for ground-truth samples. Due to the nature of the learning-based algorithm, its performance depends on several factors, such as ratio selection of the training and testing sets, clustering algorithm, network structure (from the code similarity analysis) and feature attentions (importance of different meta-information). The following subsections conduct comprehensive evaluations and discussions of these factors.

Comparing with different representation learning models. To demonstrate that our hybrid network representation learning is able to achieve better performance, we compare ANDRE with the mainstream learning methods, i.e., DOC2VEC [13] (a paragraph vectors algorithm for embedding texts and phrases in a distributed vector using neural networks) and DEEPWALK [35] (which applies language modeling to capture the topological relationships between nodes in a social network).

Figure 3.5 shows our comparison results of the three approaches. ● represents the DEEPWALK method, ● represents the DOC2VEC method, ● represents our ANDRE method. In the three subgraphs, the horizontal axis represents the ratio of the training set, which is 10% to 70%. The vertical axes in the three subgraphs represent the fractions of accuracy, macro and micro score respectively. By preserving the network structure (node-node correlation) and the co-occurrence relations between apps and their corresponding meta-info (node-word and label-word correlations), our approach performs better than the individual learning methods, i.e., DEEPWALK and DOC2VEC. In general, the accuracy values and F1 scores of the three approaches increase when the sizes of the training sets increase. The trend becomes stable and gradually reaches a convergence when the training set occupies around 70% of the total samples. ANDRE achieves up to 93% in accuracy, while DEEPWALK and DOC2VEC can only achieve 89.5% and 70.1% respectively.

Table 3.5: Comparison with different classifiers

Classifiers	Accuracy	F1 Score	Speed(Seconds)
KNN	0.8677	0.4468	225
SVM	0.8295	0.2580	256
Random Forest	0.4911	0.0491	224
Decision Tree	0.4031	0.0357	230
MLP Classifier	0.9126	0.6079	356

Comparisons using different multi-class classifiers. Given the learned representation, we compare the classification performance when applying MULTILAYER PERCEPTRON (MLP), a deep neural network classifier and conventional classification algorithms, including SVM (separating hyperplane in the feature space to maximize the interval between positive and negative samples), KNN (distance measurement between different eigenvalues), RANDOM FOREST (integrating multiple trees through Ensemble Learning) and DECISION TREE (mapping relationships between object attributes and object values through a tree structure).

To fairly compare the results of different classifiers, we keep their underlying network embeddings unchanged (i.e., by associating all meta-info for each node, using the same network structure and same representation space with the dimension K=400). As shown in Table 3.5, the accuracy of MLP classifier exceeds that of all other methods, demonstrating the recent advances in deep learning that enable the precise capture of input-output data relations correlations, i.e., the input features are connected to the neurons of the hidden layer, and the neurons of the hidden layer are then linked with the neurons of the input layer. The multi-layer perceptron layer is fully associated (the full connection means that any single neuron in the upper layer is connected to all neurons in the next layer).

Comparisons when selecting K-dimensional representation space. Table 3.6 shows the results when the number of representation dimensions K are varied from 5 to 400 in our hybrid representation learning. There are noticeable increases

Table 3.6: Performance comparison regarding different dimensions K in the representation space

Number of K	Accuracy	Macro-F1 Score	Micro-F1 Score
5	0.7230	0.2360	0.7230
10	0.8505	0.4661	0.8505
30	0.9022	0.5851	0.9022
50	0.9083	0.6115	0.9083
100	0.9151	0.6027	0.9151
200	0.9138	0.6189	0.9138
300	0.9169	0.5927	0.9169
400	0.9175	0.6282	0.9157

for accuracy, macro-F1 score and micro-F1 score when K is increased from 5 to 50. The subsequent differences are negligible especially when K is greater than 100.

All meta-info vs. permission information. In this experiment, we choose the representation dimension $K = 400$ for our MLP classifier. This experiment aims to show that of all the meta-info components, permission information makes the major contribution to the accuracy of our clustering results. All other components also contribute to the improvement in accuracy. As illustrated in Figure 3.4.3, the mean accuracy is 90% when only permission strings were used for our network embedding. The accuracy increases to 93.4% when all elements of meta-info are used. The improvement shows that the complete meta-info includes not only permission information, but also many other types of useful information, such as activity, intent, service, etc., which represents much richer text information for capturing the correlations of apps whose behaviors are similar.

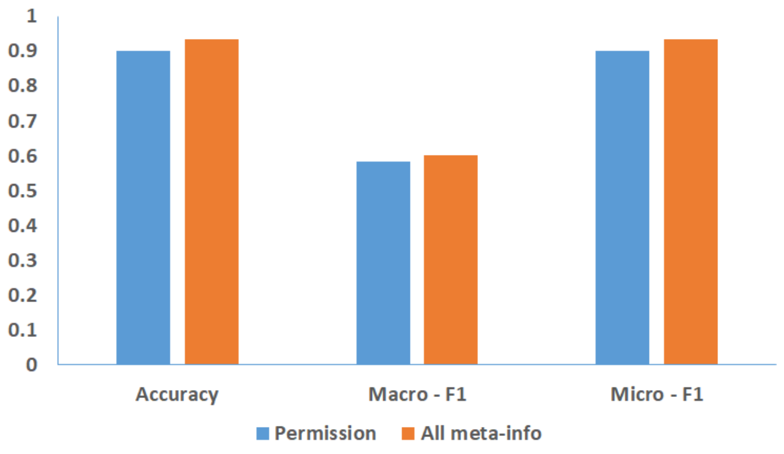


Figure 3.6: Performance comparison (permissions vs all meta-info).

Table 3.7: Results of outliers and inliers

#Weakly-labeled malware	#Malware in outlier		#Malware in inlier	
	Empty-labels	Controversial-labels	Empty-labels	Controversial-labels
# 3324	35	47	1499	1743

3.4.4 Result Analysis for Weakly-labeled Malware

When clustering the weakly-labeled malware from VirusShare, we apply our outlier-aware clustering on top of the hybrid representation learned from the constructed malware network, which consists of 5676 strongly-labeled and 3324 weakly-labeled samples.

Our outlier detection identifies that 3242 malware apps are close to existing malware families, falling into the inlier zone, and 82 apps have behaviors that are unlikely to correspond to those of the known families, thereby falling into the outlier zone.

Table 3.7 gives the inliers and outliers of the 3324 weakly-labeled malware. The number of outlier samples is relatively small, comprising only 2.5%. A major portion of the weakly-labeled malware are detected as inliers. For the 1534 empty-labeled malware (Table 3.1), 35 and 1449 are outliers and inliers respectively. Of the 1790 malware (Table 3.2) who have controversial family names (i.e., the top two most frequent family names reported by an equal number of vendors using Euphony), 47 and 1743 are outliers and inliers respectively.

Figure 3.7 shows the outlier results with 58 outlier (unknown) families clustered by anomaly detection. 47 out of 58 outliers contain only 1 malware sample, and the remaining 11 include more than 1 candidate. The largest outlier cluster contains 9 samples.

Figure 3.8 shows the inliers results, where the horizontal axis in the figure shows the family names of all malware in the inlier clusters. The vertical axis indicates the log value of the number of apps in the corresponding families. For inliers, there are 60 families out of the 176 known families. The distribution is uneven. The malware apps in the top 10 families occupy the majority (88.26%) of all the weakly-labeled malware. Regarding malware in the inlier, we observe that our method is not limited by the number of malware samples in the training set. For example, the `plankton` family ranks first in terms of the prediction result in the Drebin dataset, whereas it only ranks as the third largest family. Similarly, `Vdloader` ranks fifth according to our experiment result, while it only ranks 167th in the Drebin dataset with only 16 samples inside.

Due to the lack of ground-truths for weakly-labeled malware, manual checking is the best and only way to validate the results [88]. Following [33], we have also conducted manual inspections (costing two people for four weeks) of the malware inside the same cluster to check the similarity of their malicious behaviors (by running and looking into the source code and meta-info of the apps) to validate ANDRE's clustering results. All the malware in outliers were checked since all the outlier clusters have very small sizes. For inlier families, we randomly checked 3 apps for each family if the family contains more than 3 apps, otherwise, all the apps within the family were checked. This process includes running the app (in a virtual Android environment) and checking the meta-info, vendors' raw labels and malicious code (decompiled by `dex2jar`) to understand their familial behaviors. We will select and illustrate in detail the representative apps within 3 different clusters in the inlier set in the following subsection.

3.4.5 Case Studies for Weakly-labeled Malware in Inlier

This section conducts in-depth case studies to demonstrate representative inlier weakly-labeled malware clustered by ANDRE. Since every malware has a unique hash value (MD5), we will refer to their hash values in the following studies.

Inlier case study 1 - YzchBgserv family. Malware "327a0387a3114ab4e4d 18a81ad9fca8b", which is a malware with an empty label being processed by EUPHONY, is clustered into the YzchBgserv family by ANDRE based on our hybrid representation. The app has similar malicious behaviors to other apps in YzchBgserv, which is essentially a type of ransomware. The app runs as a background thread which is automatically triggered when the app starts. It first fetches a target service provider number (component info) from a remote server and then sends an SMS message (permission info) to the number, incurring a charge on the user's phone bill. The SMS messages (code info) sent will always start with a string "YZHC".

Inlier case study 2 – DroidKungFu family. Malware "00307253cd9ad3d11 20df85beb473801" which is unable to be clustered by EUPHONY, is clustered into the DroidKungFu family by ANDRE. We manually checked the app whereas observing that such malware infects a self-defined service and receiver (code-info), which can be automatically launched without user interaction. Once the service gets started, it will collect three kinds of user information on the infected device, including the IMEI number, phone model, and the Android OS version (meta-info). These behaviors are typical symptoms of being infected by the Droidkungfu family.

Inlier case study 3 – Fakeupdates family. Malware "005054fdf485fdbbada 461bd7824cfb2", whose top two families are reported by an equal number of vendors ("fakeupdates":1, "igexin":1). After manual validation, we established that the `plankton.device.android.service` package is its source of malice. By looking at several other peers within the same cluster, we find that they all share this package (code-info) with an identical code segment to start the malicious service (i.e., AndroidMDKService. initMDK ()).

Table 3.8: Suspicious permissions used by apps in Outlier-1

Permission Strings
ACCESS_COARSE_LOCATION
ACCESS_FINE_LOCATION
ACCESS_FINE_LOCATION
INTERNET
READ_PHONE_STATE
SYSTEM_ALERT_WINDOW
WRITE_EXTERNAL_STORAGE
ACCESS_DOWNLOAD_MANAGER
DOWNLOAD_WITHOUT_NOTIFICATION
WRITE_SETTINGS

3.4.6 Case Studies for Weakly-labeled Malware in Outlier

We also performed a manual inspection of outliers whose malicious behaviours are different from the known families in the network. As per our discussions in Section 3.3.4, the results of our outlier detection often rely on the availability of the existing known families in the network, which may not cover all emerging malware families. The following case studies report the outlier clusters, inside which an app does not conduct behaviors that are similar to those of the existing ground-truths (e.g., Drebin).

Outlier-1 case study. This cluster, containing 9 malware, is the largest of all outlier clusters. Our manual inspection reveals that the apps in this cluster share the same risky permission combinations (9 suspicious permissions in total, as shown in Table 3.8) to trigger similar malicious behaviors, which induce users to click an inbound link address to install a trojan package that modifies the app's configuration files. For example, the malware candidate "ee93c3eefb81151eca7346db74d613c" has the top two family names "autoins" and "letang" both reported by 2 vendors using Euphony. These two family names are new and do not appear in our ground-truths.

Outlier-2 case study. This cluster contains 4 malware sharing similar malicious behaviors as triada [103], a recent malware family, which does not belong to any of the existing families in the network. The malware in this cluster first collects sensitive information including Android OS versions and the list of installed applications. Then it sends this information to the command and control (C&C) server [104].

Outlier-3 case study. This cluster also contains 4 malware in total. The malice of the malware in this cluster starts from their in-app advertisements. In particular, when a user clicks a pre-defined advertisement link, the malicious program will display as many ads as possible to the user. These malware also have a risky combination of permissions using CHANGE_WIFI_STATE and CHANGE_NETWORK_STATE whose intention is to switch the Wifi and network status respectively after deriving their current status.

3.5 Discussions and Limitations

First, like all learning-based approach, our approach requires samples for training our model for precise clustering of weakly-labeled malware. The more samples we have, the more precise and robust our clustering model will be.

Second, this approach relies on a whitelist [92] to exclude common Android

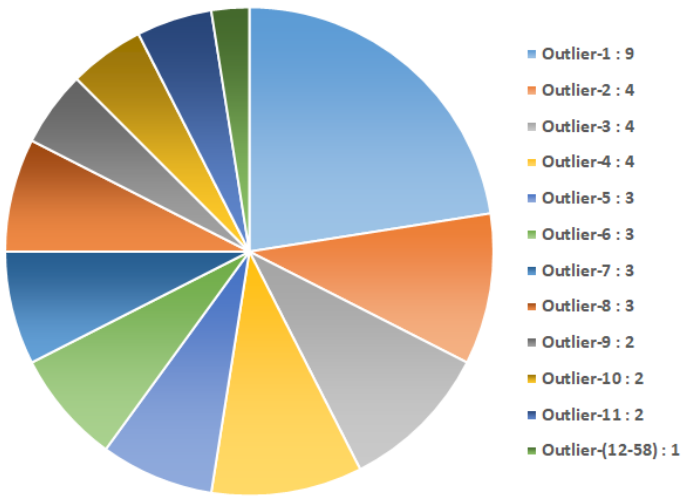


Figure 3.7: Weakly-labeled malware detected as outliers (unknown families in the network).

application third-party libraries, which may potentially lead to false positives or negatives, if the whitelist is incomplete for emerging APIs. Nevertheless, whitelist is still a standard way for reducing noise in code similarity analysis. In addition, whitelist also provides a flexible way for adding any new APIs which are safe.

Third, regarding the calculation of the similarity distance between two Android applications, our method uses Jaccard for measuring the similarity distances, which achieves good results for code analysis. However, other methods, such as Cosine [105], can also be used as an alternative approach to similarity analysis.

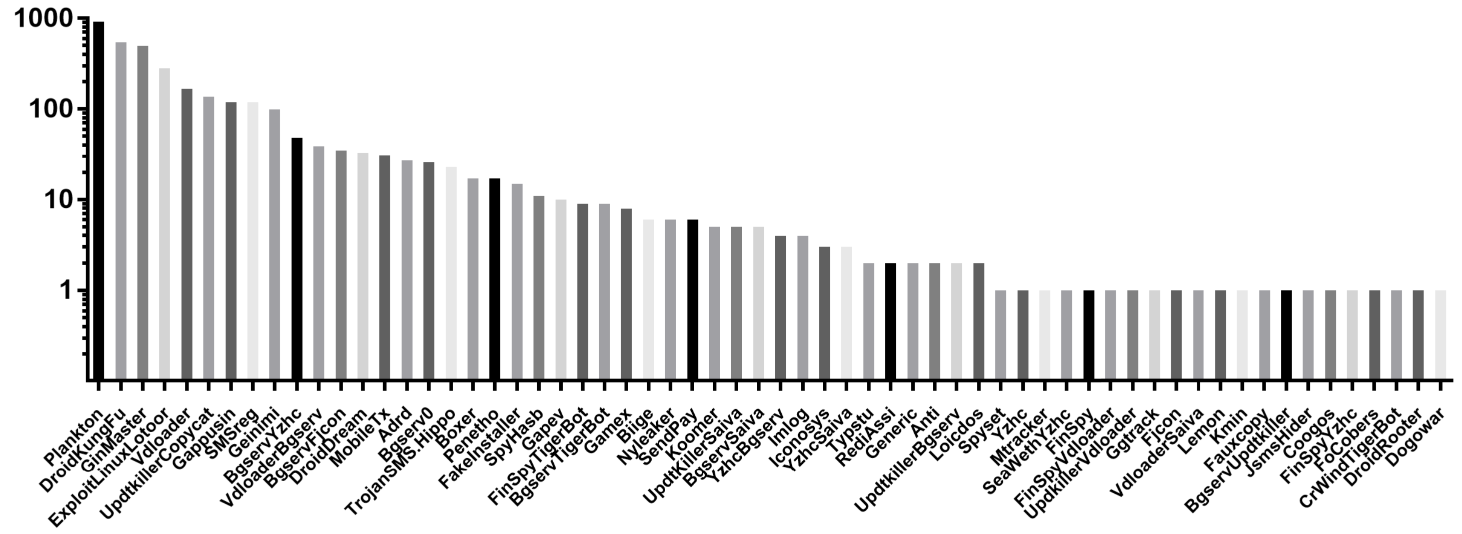


Figure 3.8: Weakly-labeled malware detected as inliers (known families in the network).

Fourth, ANDRE aims to develop a representation learning framework for clustering weakly-labeled malware. When comparing the similarity between Android apps, our method does not focus on obfuscation or deobfuscation, which may lead to imprecise results if malicious apps are obfuscated. Investigating obfuscated app is an orthogonal but an interesting future topic.

Finally, when calculating the similarity between two apps, we made a pairwise comparison for all Android malware. Applying parallel computing or enable similarity comparison between apps using multiple threads can reduce the code analysis overhead and save comparison time.

CHAPTER 4

Investigating Code Deobfuscations on Detecting Obfuscated Piggybacked Apps

4.1 Introduction

As the world's most popular mobile platform, the Android operating system (OS) impacts significantly on many people's daily life, thus the interest in its security and security enhancement is gaining increasing momentum in areas from academia to industry [79, 106]. Android apps are written in Java programming language, making them easy to be reversed by development tools. For example, the Apktool is often used to disassemble executable code and decode resource files of Android apps [107]. Since Android allows self-signed certificate apps, once an app is disassembled and decoded, its code and resource files can be modified, resigned, and repackaged as a new app. Worse, users feel free to install these unofficially released apps, resulting in potential security issues.

Malware developers usually unpack benign and preferably popular apps for piggybacking code fragments with malicious behaviours, as illustrated in Figure 4.1. These apps are referred to as piggybacked apps, and are widely available in real-world markets, which is seriously threatening user security and destroying the reputation

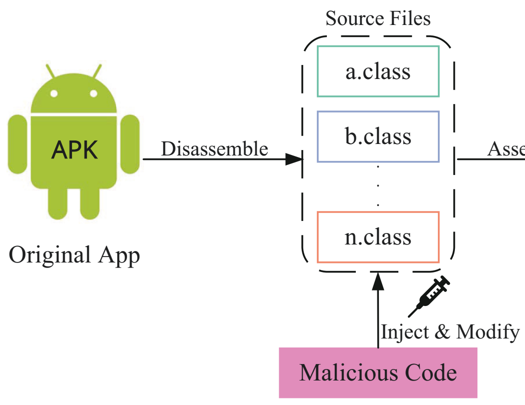


Figure 4.1: Illustration of Android piggybacked malware.

of the developers of the original apps. Zhou *et at.* [47] pointed out that 86% of the 1,260 malicious application samples examined are piggybacked malicious apps. To protect users against these malicious apps, various anti-malware detectors, such as VirusTotal and SimiDroid, have been released to detect malware [108].

Unfortunately, piggybacked app developers normally use code obfuscations to evade such detection. Code obfuscation is a reorganization and reprocessing technology used on previously released programs [40]. An obfuscation technique transforms the original program into a new one while maintaining its functionality, but the obfuscation code makes the decompilation difficult to figure out the true semantics of the original program. Recent studies have shown that code obfuscations used in piggybacked apps have caused a significant decrease in the ability to accurately detect Android malware [75, 109].

To overcome the challenge of detecting obfuscated Android malware, researchers have developed deobfuscation tools to identify obfuscated codes injected into apps by malware writers. For example, Simplify [20], developed by Caleb Fenton, uses a virtual machine sandbox for executing an app to understand its behavior. Then, it tries to optimize the code so that the decompiled code is simplified and easier for humans to understand. Different from the work by Fenton, Bichsel *et al.* [15] developed Deguard, a statistical deobfuscation tool for Android. Deguard phrases the layout deobfuscation problem of Android apps as a structured prediction in a probabilistic graphical model for identifiers that are based on the occurrence of names.

However, there is little empirical evidence on the impact of code deobfuscations on detecting obfuscated Android piggybacked apps. Such empirical knowledge can provide useful insights for researchers and security engineers to better understand the influences of obfuscation strategies and deobfuscation tools on Android malware detectors. Thus, such evidence can guide the design of resilient and effective Android malware detectors. Therefore, in this chapter we perform a large-scale empirical study of the impact of code deobfuscations on detecting obfuscated Android piggybacked apps.

Figure 4.2 shows an overview of our empirical study. First, we generated obfuscated piggybacked apps using 1,399 pairs of original piggybacked apps from [110] by utilizing different obfuscation strategies. The generated obfuscated apps were later deobfuscated to generate deobfuscated piggybacked apps. To present a thorough analysis, we used three types of Android malware detectors, including commercial anti-malware products, machine learning-based detectors, and similarity-based detectors, as the targeted systems for detecting the generated obfuscated/deobfuscated piggybacked apps. Finally, having used 10 different strategies, we analyzed the precision of these detectors to understand the impact of code deobfuscations.

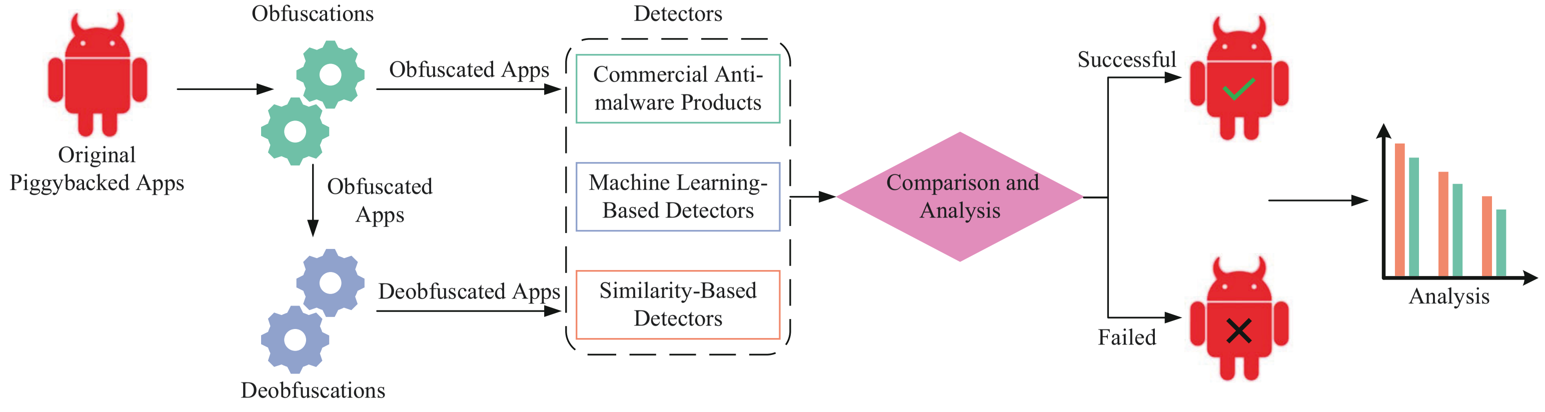


Figure 4.2: Overview of our empirical study.

4.2 Research Methodology

4.2.1 Dataset and App Generation

Original Apps. We use the dataset, i.e., 1,497 pairs of original/piggybacked Android apps, which were collected from the work conducted by Li *et al.* [110]. In the work, the authors first sent all apps to VirusTotal to collect their associated anti-virus scanning reports. Then, based on the results of VirusTotal, they classified the set of apps into two subsets: one containing only benign apps and the other containing only malicious apps. Then they compared all the malicious apps and the benign apps according to the identity packages. If the pair of apps that were compared had different authors, they continued the comparison to establish whether the SDK version is the same. Finally, they collected the dataset of 1,497 app pairs, where each pair include an original benign app and a piggybacked app with a malicious payload. According to the provided hash values, we downloaded all 1,497 pairs of original/piggybacked Android apps.

Obfuscated Piggybacked Apps. Among the 1,497 app pairs, there are some duplicates after our manual inspection. The dataset has several benign apps corresponding to the same piggybacked app. After filtering these duplicate piggybacked apps, we obtained validated 1,399 pairs for our study. In order to understand the impact of each obfuscation strategy, we used AVPASS to obfuscate the validated 1,399 apps pairs with 10 obfuscation strategies (Table 4.1). Table 4.2 shows the number of obfuscated piggybacked apps generated for each obfuscation strategy. Note that the numbers of obfuscated piggybacked apps can be different for each strategy, since not all the 1,399 piggybacked apps can be successfully obfuscated by the obfuscation strategies. For example, only 444 piggybacked apps were successfully obfuscated using BYTECODE.

Deobfuscated Piggybacked Apps. To create deobfuscated piggybacked apps, we leveraged two general Android deobfuscator, i.e., Simplify [20] and Deguard [15].

Simplify. The design of Simplify is inspired by dex-oracle, in which the Dalvik virtual machine is simulated, and the code is decompiled once executed [20]. Af-

Table 4.1: Obfuscation Strategies

Type	**Strategy**	**Description**
CFF	API_REFLECTION	Hides all APIs by Java reflections.
	API_INS	Inserts invalid APIs between existing APIs.
IJC	BYTECODE	Inserts useless codes in the source code, such as defining a useless method, passing the class variable in, then not processing or shifting the useless codes, which looks like complicated codes, but actually same function.
IO	VARIABLE	Renames the variable name in the source code, then replaces it with a meaningless identifier, making it harder to crack this analysis.
	PCM	Obfuscate the package name in the same way as the variable name.
	BENIGN_CLASS	Obfuscate the class name in the same way as the variable name.
	RESOURCE_IMAGE	Modifies directly at the source level, then replaces the code and renames "icon.png" to "a.png", lastly hands it over to Android for compilation.
	RESOURCE_XML	Modifies directly at the source level, then replaces the code and "R.string.name" in xml file with "R.string.a", and hands it over to Android for compilation.
SO	STRING	Encrypts the string locally, then hardcodes the ciphertext into it. Lastly, decrypts it at runtime.
	BEN_PERMISSION	Encrypts the permission locally, hardcodes the ciphertext into it, and decrypts it at runtime.

ter learning the function, the decompiled code is simplified into a form that the analyst can understand. Simplify includes two modules, i.e., Smalivm and Simplify. Smalivm is the simulator module of the Dalvik virtual machine and is mainly used for executing Dalvik virtual machine, based on the input smali file, returns all possible execution paths. The simplify module is the main module used to solve the obfuscation, which is primarily based on the analysis results of Smalivm. It simplifies the obfuscated decompiled code and generates the easy-to-understand decompiled code. We used Simplify to generate the deobfuscated piggybacked apps, using the obfuscated apps (Table 4.2). The number of deobfuscated piggybacked apps by Simplify is shown in Table 4.3.

Deguard. Deguard is a new system for statistical deobfuscated Android APKs [15]. It deobfuscates an APK with large-scale learning and then summarizes a probability model to identify the code through its probability model. Using these models, Deguard restores important information in the Android APK, including method and class names, as well as third-party libraries. Deguard then reveals string decoders and classes that handle sensitive data in Android malware. The process is divided into three steps: (1) Generating a dependency graph, where each node represents the element to be renamed, and each line represents a dependency; (2) Exporting restriction rules, which guarantees the APK being replied is a normal APK and keeping the same semantics as the original APK; (3) Predicting and recovering the original name of an obfuscated element according to the weighting provided by the probability model.

We wrote a python script with selenium [111] in order to use Deguard. Python script automatically opens the official Deguard website and then uploads the piggybacked apps. Once the deobfuscation process has taken place, it automatically downloads the generated deobfuscated apps, which has been named with a suffix, i.e., the hash value of the original app. Because the Deguard tool is not open-source and its website does not support batch operation to handle a set of apps. Uploading each individual piggybacked app to its website is very slow and time consuming. Therefore, for each strategy, we randomly select 50 obfuscated apps to generate their deobfuscated apps, to make our evaluation as fair as possible.

Table 4.2: Number of Obfuscated Apps

Strategy	**Number**	**Strategy**	**Number**
STRING	1,130	BE_CLASS	1,372
VARIABLE	1,148	API_INS	933
PCM	1,311	BEN_PER	1,369
BYTECODE	444	API_REF	1,282
RESOURCE_IMAGE	1,370	RESOURCE_XML	1,370

Table 4.3: Number of Deobfuscated Apps (Simplify)

Strategy	**Number**	**Strategy**	**Number**
STRING	1,130	BE_CLASS	1,370
VARIABLE	1,148	API_INS	933
PCM	1,311	BEN_PER	1,369
BYTECODE	444	API_REF	933
RESOURCE_IMAGE	1,370	RESOURCE_XML	1,370

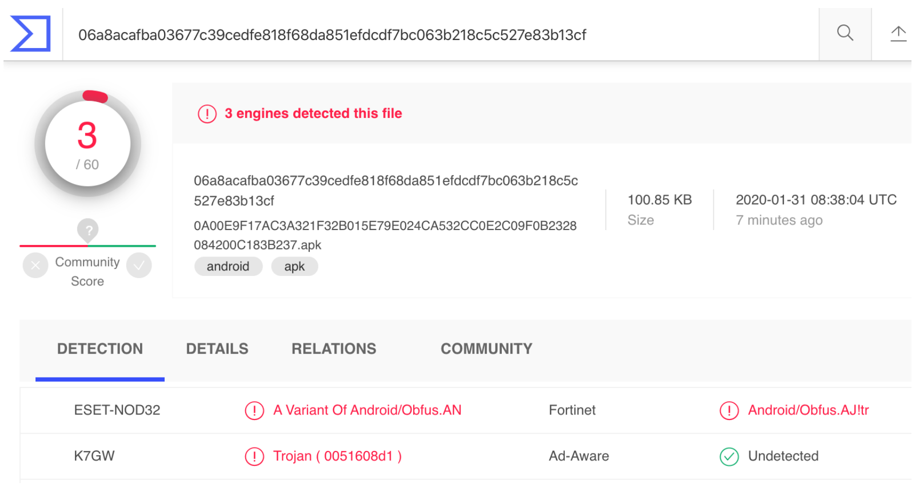
4.2.2 Targeted Systems

To study the impact of code deobfuscation on detecting obfuscated piggybacked apps, we selected three types of Android malware detectors, including commercial anti-malware products, machine learning-based detectors, and similarity-based detectors.

Commercial Anti-malware Products. As shown in Figure 4.3, VirusTotal is a website (founded in 2004) which provides free suspicious file analysis services. Different from traditional anti-virus software, VirusTotal scans files through multiple commercially available anti-virus engines and then uses a variety of anti-virus engines to inspect the files uploaded to determine if the files are infected by viruses, worms, trojans or other types of malware. This greatly reduces the chances of the anti-virus software, either missing or not detecting a virus. Its detection rate is significantly better than when a single anti-virus product is used. Whilst no anti-virus software is 100% safe, the VirusTotal test results are more comprehensive and much more likely to be correct than if a single anti-virus engine was used. In addition, the compressed file structure (compressed file), file type - MD5, SHA1, SHA256 - can also be analyzed. In this study, after obfuscation and deobfuscation, if the number of anti-virus engines in VirusTotal that detect malware variants is greater than that of detecting original malware, the detection precision is considered to be increased, otherwise, the detection precision is considered to be decreased.

Machine Learning-Based Detectors. Drebin is a lightweight method to detect Android malware [22]. It automatically infers the detection mode and directly identifies malware on mobile phones. First, Drebin collects as many features as possible from the application with static analysis. The collection of features mainly includes four sets of features from the manifest file (i.e., permissions, hardware components, APP components, and filtered intents) and four sets of features from disassembled code (i.e., restricted API calls, user permissions, suspicious API calls, and network addresses). These features are then organized into a set of strings, embedded in the vector space, and geometrically analyzed the patterns and combinations of these features geometrically analyzed. Finally, the embedded feature set is used to identify Android malware. Drebin learns a linear SVM classifier to discriminate between benign and malicious apps.

CSBD is a method containing a python-based reimplementation of the Android malware detection [112]. CSBD uses control flow graph (CFG) signatures of methods in Android apps to detect malicious apps. Firstly, CSBD performs static analysis of the Android app's bytecode to extract a representation of the program's CFG, which is then expressed as character strings so that the similarity between Android apps


06a8acafba03677c39cedfe818f68da851efdcdf7bc063b218c5c527e83b13cf

3 / 60

Community Score

3 engines detected this file

06a8acafba03677c39cedfe818f68da851efdcdf7bc063b218c5c527e83b13cf
0A00E9F17AC3A321F32B015E79E024CA532CC0E2C09F0B2328084200C183B237.apk
android apk

100.85 KB
Size

2020-01-31 08:38:04 UTC
7 minutes ago

DETECTION DETAILS RELATIONS COMMUNITY

ESET-NOD32	A Variant Of Android/Obfus.AN	Fortinet	Android/Obfus.AJ!tr
K7GW	Trojan (0051608d1)	Ad-Aware	Undetected

Figure 4.3: Example of VirusTotal.

can be established. This derived string representation of the CFG is an abstraction of the app's code, which retains information about the structure of the code but discards low-level details such as variable names or register numbers. Next, after having the abstract representation of an Android app's CFG, CSBD collects all basic blocks that compose and refer them as the features of the app. A basic block is taken as a sequence of instructions with only one entry point and one exit point, which represents the smallest piece of the app. By leveraging machine learning techniques and learning from the training dataset, the model exposes those basic blocks that statistically appear more frequently in Android malware.

Similarity-Based Detector. Androguard is an open-source Android app analysis tool [113]. It implements a common similarity calculation method based on the original code at the method level. The feature extraction in Androguard is designed to generate an abstract representation of the method signature and statement. The latter represents the type from the statement (e.g., the if-statement, invoke statement) rather than the exact statement string. In addition, it also extracts constants, for example, numbers and strings as comparison features. Androguard then leverages normalized compression distance (NCD) to compute the similarity distance between two different methods. This is done by using state-of-the-art compressors to calculate the similarity between the two methods, rather than comparing all the statements in a given method.

SimiDroid detects similar Android apps and explaining the identified similarities at different levels [114] (i.e., similarities at method level, component level, and resource level). In our study, as Androguard has compared the similarity of an app pair at the method level, we only use SimiDroid to compare APK file similarities at the component and resource levels.

We computed the similarity score of the given two apps (e.g., app1 and app2) using Equation 4.1. Given a pre-defined threshold, which can be computed based on a set of known piggybacked pairs, it is then possible to conclude with confidence that the given two apps are similar.

$$similarity = max(\frac{identical}{total - new}, \frac{identical}{total - deleted}), \tag{4.1}$$

where *identical* denotes that when a given key/value entry is matched exactly the same in both maps; *similar* is that when a given key/value entry slightly varies slightly from one app to the other in a pair and more specifically when the key is the same but values differ; *new* represents that when a given key/value entry exists only in map 2 but not in map 1; *deleted* denotes that when a given entry existed in map 1, but is no longer found in map 2. In addition, $total = identical + similar + new + deleted$.

Table 4.4: Precision of Detecting Obfuscated Piggybacked Apps by Different Detectors

Type	**Strategy**	**VirusTotal**	**Drebin**	**CSBD**	**Androguard**	**SimiDroid-C**	**SimiDroid-R**	**Average**	**STDEV**
CFF	API_REFLECTION	54.7%	82.8%	37.6%	28.2%	100.0%	57.9%	60.2%	0.271
	API_INS	70.9%	77.5%	44.9%	35.7%	100.0%	58.6%	64.6%	0.233
IJC	BYTECODE	51.9%	33.3%	13.0%	61.8%	100.0%	38.9%	49.8%	0.297
IO	VARIABLE	88.5%	75.6%	81.5%	99.0%	100.0%	55.5%	83.3%	0.166
	PCM	77.3%	78.6%	82.0%	98.7%	3.6%	56.4%	66.1%	0.335
	BENIGN_CLASS	89.4%	72.3%	34.1%	58.9%	100.0%	57.0%	68.6%	0.238
	RESOURCE_IMAGE	64.2%	78.4%	73.0%	99.0%	100.0%	57.0%	78.6%	0.177
	RESOURCE_XML	76.2%	78.4%	77.9%	99.0%	100.0%	57.0%	81.4%	0.161
SO	STRING	101.7%	77.6%	51.6%	35.1%	100.0%	57.2%	70.5%	0.271
	BEN_PERMISSION	74.6%	81.0%	79.0%	99.0%	100.0%	57.0%	81.7%	0.161

Note. STDEV means the standard deviation of precision of each strategy to different detectors.

4.2.3 Experiment Designs

We used the VirusTotal service to scan the generated obfuscated and deobfuscated piggybacked apps using anti-malware products by uploading the APK files to VirusTotal. For each uploaded app, VirusTotal returned a unique report. By analyzing the statistical results, we were able to obtain the detecting results of various commercial anti-virus products of both the obfuscated and deobfuscated piggybacked apps.

Additionally, we used two aforementioned machine learning-based detectors (i.e., Drebin and CSBD) to scan the obfuscated and deobfuscated apps. The two machine learning-based detectors were trained by the original dataset, i.e., 1,399 pairs of original/piggybacked apps. After the models were trained, they were then used to detect the generated obfuscated and deobfuscated datasets.

Furthermore, we utilized the two similarity-based detectors, including Androguard and SimiDroid, to detect the generated piggybacked apps. Androguard was used to compare original piggybacked malware at the method level and its corresponding malware that had been obfuscated by various strategies. In addition, SimiDroid was used at both the component level and the resource file level to compare the corresponding malware pairs.

The evaluation metric we used here are the precision of the malware detector. The precision is defined as $Precision = TP/(TP + FP)$, where TP is the number of correctly classified malware and FP is the number of benign apps which are predicted as malware. The reason why we choose precision as the evaluation metric is that if the detection precision of malware detectors is improved after deobfuscation, it means that the readability of the code is improved for the detectors.

4.3 Results and Analysis

Our empirical study aims to answer the following three research questions (RQs):

- **RQ1.** How is the precision of Android anti-malware detectors impacted by obfuscation strategies?

- **RQ2.** How is the precision of Android anti-malware detectors impacted by deobfuscation strategies?

- **RQ3.** How do different deobfuscation tools impact the precision of Android anti-malware detectors?

4.3.1 RQ1. Impact of Code Obfuscations

The precision of detecting obfuscated piggybacked apps by different detectors is shown in Table 4.4. Note that the precision of detecting original piggybacked apps (i.e., without code obfuscation) by each detector is 100%.

Commercial Anti-malware Products. Table 4.4 presents that if piggybacked apps are obfuscated, the detection precision of commercial anti-virus software will be greatly reduced. Of the 10 obfuscation strategies used, BYTECODE has the most severe impact on anti-virus products, since the precision of detection decreases the most. This is followed by API_REFLECTION, RESOURCE_IMAGE, and API_INS. The least severely affective obfuscation strategies on the anti-virus products of VirusTotal are VARIABLE and BENIGN_CLASS. Note that after being obfuscated by STRING, the detection precision of VirusTotal did not fall but slightly increase. STRING is to obfuscate string in the original code of an APK file, which can be speculated. Most anti-virus products already have a mature response for obfuscation strategy for strings, making them sensitive to string obfuscation.

Machine Learning-Based Detectors. The obfuscation strategies also reduce the precision of the Drebin detector. However, apart from BYTECODE, the influences of other obfuscation strategies on the Drebin detector are similar. BYTECODE is a very successful evade strategy for this machine learning-based detector Drebin. BYTECODE changes the package name of Android piggybacked apps.

Although CSBD is also a machine learning-based detector, it uses a completely different feature (i.e., CFG of the APK file) than Drebin. As shown in Table 4.4, the impacts of different obfuscation strategies for CSBD detector on obfuscated Android piggybacked apps for the CSBD detector are more pronounced than Drebin. In particular, STRING, BYTECODE, BENIGN_CLASS, API_INS, and API_REFLECTION significantly changes to CSBD's precision. However, the impact of other strategies on CSBD is similar to their impact on Drebin.

Table 4.5: Precision of Detecting Deobfuscated Piggybacked Apps by Different Detectors (Simplify)

Type	Strategy	VirusTotal	Drebin	CSBD	Androguard	SimiDroid-C	SimiDroid-R	Average	STDEV
CFF	API_REFLECTION	75.9% ↑	77.9% ↓	43.4% ↑	28.1% ↓	100.0% −	58.3% ↑	63.9% ↑	0.259
	API_INS	77.0% ↑	77.9% ↑	43.2% ↓	35.6% ↓	100.0% −	58.6% −	65.4% ↑	0.241
IJC	BYTECODE	59.0% ↑	63.3% ↑	33.0% ↑	61.8% −	100.0% −	38.9% −	59.3% ↑	0.235
IO	VARIABLE	75.8% ↓	76.3% ↑	69.4% ↓	99.9% ↑	100.0% −	55.5% −	79.5% ↓	0.175
	PCM	64.3% ↓	78.6% −	75.3% ↓	98.6% ↓	3.6% −	56.3% ↓	62.8% ↓	0.323
	BENIGN_CLASS	70.2% ↓	79.0% ↑	65.5% ↑	58.9% −	100.0% −	57.0% −	71.0% ↑	0.159
	RESOURCE_IMAGE	72.1% ↑	79.0% ↑	70.1% ↓	99.9% ↑	100.0% −	57.0% −	79.7% ↑	0.172
	RESOURCE_XML	73.2% ↓	79.0% ↑	74.5% ↓	99.9% ↑	100.0% −	57.0% −	80.6% ↓	0.167
SO	STRING	72.7% ↓	78.4% ↑	47.9% ↓	35.1% −	100.0% −	57.2% −	65.2% ↓	0.232
	BEN_PERMISSION	71.2% ↓	81.6% ↑	69.8% ↓	99.9% ↑	100.0% −	57.0% −	79.9% ↓	0.173

Note. "↑", "↓", and "−" after each cell value denote that the detection precision has been improved, decreased, or continued after deobfuscation compared to detecting obfuscated piggybacked apps. *STDEV* means the standard deviation of precision of each strategy to different detectors.

Table 4.6: Precision of Detecting Deobfuscated Piggybacked Apps by Different Detectors (Deguard)

Type	Strategy	VirusTotal	Drebin	CSBD	Androguard	SimiDroid-C	SimiDroid-R	Average	STDEV
CFF	API_REFLECTION	66.6% ↑	52.9% ↓	66.6% ↑	53.9% ↑	100.0% −	97.5% ↑	72.9% ↑	0.208
	API_INS	73.9% ↑	52.9% ↓	0.0% ↓	67.3% ↑	100.0% −	97.6% ↑	65.3% ↑	0.367
IJC	BYTECODE	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*
IO	VARIABLE	55.9% ↓	68.6% ↓	41.1% ↓	71.2% ↓	100.0% −	97.8% ↑	72.5% ↓	0.230
	PCM	46.8% ↓	74.5% ↓	46.6% ↓	68.7% ↓	100.0% ↑	98.2% ↑	72.4% ↑	0.235
	BENIGN_CLASS	49.7% ↓	50.9% ↓	25.5% ↓	64.7% ↑	100.0% −	98.2% ↑	64.8% ↓	0.293
	RESOURCE_IMAGE	54.9% ↓	60.7% ↓	36.0% ↓	71.2% ↓	100.0% −	97.8% ↑	70.1% ↓	0.250
	RESOURCE_XML	54.5% ↓	74.5% ↓	46.1% ↓	70.5% ↓	100.0% −	98.1% ↑	73.9% ↓	0.220
SO	STRING	88.9% ↓	60.7% ↓	12.0% ↓	89.1% ↑	100.0% −	97.6% ↑	74.7% ↑	0.337
	BEN_PERMISSION	50.2% ↓	80.3% ↓	53.1% ↓	70.6% ↓	100.0% −	98.2% ↑	75.5% ↓	0.214

Note. "↑", "↓", and "−" after each cell value denote that the detection precision has been improved, decreased, or continued after deobfuscation compared to detecting obfuscated piggybacked apps. For BYTECODE strategy, Deguard cannot generate deobfuscated application, so it is displayed as *NaN* in the table. *STDEV* means the standard deviation of precision of each strategy to different detectors.

Similarity-Based Detector. Androguard determines the similarity between two apps based on the method level. As shown in Table 4.4, the impact of different strategies on Androguard is significantly different from the other detectors used in this study. VARIABLE, PCM, RESOURCE_IMAGE, RESOURCE_XML, and BEN_PERMISSION have little effect on Androguard's detection performance, which means Androguard believes that the similarity score of the piggyback malware pair is close to one. Unlike the aforementioned strategies, STRING, BYTECODE, BENIGN_CLASS, API_INS, and API_REFLECTION have a huge impact on the performance of the Androguard. This is because these strategies are mainly changed at the method level, which is sensitive to the Androguard.

The similarity between the two apps is determined by comparing the component level methods. It can be observed from Table 4.4 that SimiDroid's resilience is excellent against all the obfuscation strategies except PCM. This is because PCM is to obfuscate the name of the component in the source code.

The comparison of resources is used to determine the similarity between the two apps' APK files. Table 4.4 shows that all the obfuscation strategies drop the detection performance of this detector. Most obfuscation strategies will drop their performance from 100% to 55%, while BYTECODE falls even more to 40%.

Finding #1: *If the attackers apply obfuscation technologies to a malware, all the antivirus products' detection capability against it will reduce from 20% to 50%. In particular, BYTECODE has the most significant impact on the detection precision, while BEN_PERMISSION has the least impact.*

4.3.2 RQ2. Impact of Code Deobfuscations

Table 4.5 and Table 4.6 show the precision of detecting deobfuscated piggybacked apps by different detectors. Note that the values in the row of BYTECODE of Table 4.6 are *NaN*, since Deguard can not deobfuscate obfuscated apps using BYTECODE strategy.

Commercial Anti-malware Products. Table 4.5 shows that the detection precision after deobfuscation is worse than that of detecting the obfuscated piggybacked apps by six strategies (i.e., VARIABLE, PCM, BENIGN_CLASS, RESOURCE_XML, STRING, and BNE_PERMISSION). Identifier-renaming strategies (e.g., STRING, VARIABLE, PCM, and BENIGN_CLASS) can decrease the detection precision after deobfuscations. These strategies change the previous names without knowledge of the original names, so the renaming deobfuscation is very difficult. However, deobfuscators can convert invisible characters into visible characters or long strings to short strings. As a result, a simple string replacement makes deobfuscated piggybacked apps look less suspicious, making them more likely to evade malware detectors.

On the contrary, four strategies (i.e., BYTECODE, RESOURCE_IMAGE, API_INS, and API_REFLECTION) by using Simplify can increase the precision of VirusTotal for detecting deobfuscated piggybacked apps by 7.2%, 7.8%, 6.1%, and 21.2%, respectively, compared to detecting obfuscated apps. Because obfuscated Android piggybacked apps after deobfuscation with these strategies can restore its malicious behaviour, thereby increasing the detection precision.

Moreover, for the deobfuscation tool Deguard (Table 4.6), only two deobfuscation strategies (i.e., API_INS and API_REFLECTION) can increase the precision of detecting deobfuscated piggybacked apps. The rest of strategies make the detection precision worse than that of detecting obfuscated malware.

Finding #2: *The precision of commercial anti-malware products for detecting deobfuscated piggybacked apps can be improved by engaging CFF-related strategies (i.e., API_REFLECTION and API_INS) for both Simplify and Deguard. However, after deobfuscations with IO-related (except RESOURCE_IMAGE for Simplify) and SO-related strategies, the precision of detecting debofuscated piggybacked apps would be decreased.*

Machine Learning-Based Detectors. After deobfuscation tool (i.e., Simplify) using all the strategies (except API_REFLECTION and PCM), the precision of detecting deobfuscated piggybacked apps can be slightly improved compared to detecting obfuscated malware. In particular, BYTECODE strategy has a significant impact on the precision of Drebin for detecting deobfuscated piggybacked apps, i.e., the detection precision has increased from 33.3% to more than 63.3%. However, after deobfuscation using Deguard, the precision would be decreased.

Drebin's mechanism is to collect a large amount of information in the APK as features. As a result, slight modifications of the code in the APK file, such as simple renaming, are not very sensitive to Drebin. Therefore, the more features are taken for a machine learning-based detector, the better the detection precision and obfuscation resilience capability.

Finding #3: *Simplify helps improve the precision of Drebin on detecting piggybacked apps obfuscated with all the strategies except API_REFLECTION and PCM, while deobfuscations using Deguard decreases the detection precision.*

For Simplify, BYTECODE, BENIGN_CLASS, and API_REFLECTION can significantly improve the precision of CSBD for detecting deobfuscated piggybacked apps, compared to when used on obfuscated piggybacked apps. In addition, for Deguard, only API_REFLECTION strategy can improve the detection precision. For all other strategies, the detection results are worse than that of detecting obfuscated piggybacked apps.

Finding #4: *Simplify with BYTECODE, BENIGN_CLASS, and API_REFL ECTION helps improve the precision of CSBD when detecting deobfuscated piggybacked apps. For Deguard, only API-related strategy (API_REFLECTION) improves the precision of CSBD, while the rest strategies decrease the precision.*

Similarity-Based Detector. For the deobfuscation tool Simplify, the precision of detecting deobfuscated piggybacked apps is similar to the detection of obfuscated piggybacked apps. Therefore, we can conclude that code deobfuscation has no significant impact on the similarity-based detector, which uses the method level feature. However, for Deguard, CFF-related strategies (i.e., API_REFLECTION and API_INS) significantly improve the precision of detecting deobfuscated malware compared with detecting obfuscated malware.

Finding #5: *After deobfuscation using Simplify with all strategies, the precision of Androguard for detecting deobfuscated piggybakced malware is similar to detecting obfuscated piggybacked apps. However, Deobfuscation using Deguard with CFF-related strategies (i.e., API_REFLECTION and API_INS) can significantly improve the detection precision of Androguard for detecting deobfuscated piggybacked apps.*

For Simplify, except PCM, the resilience of SimiDroid is excellent against almost all the deobfuscation strategies. Similar to the method level similarity-based detector, the deobfuscation strategy has little effect on the component level similarity-based detector. Moreover, the detection precision of the similarity-based detector is not affected by Deguard's deobfuscation, especially PCM, which Simplify deobfuscation.

With deobfuscation using Simplify, there is a slight change in the detection precision. However, using Deguard for deobfuscation, the precision of SimiDroid-R for detecting deobfuscated malware is greatly improved when using Deguard with all the strategies.

Finding #6: *Code deobfuscation has no significant impact on a similarity-based malware detector using a component-level feature to detect deobfuscated piggybacked apps. In addition, a deobfuscation tool using Deguard can significantly improve the precision of a similarity-based malware detector using a resource-level feature to detect deobfuscated malware.*

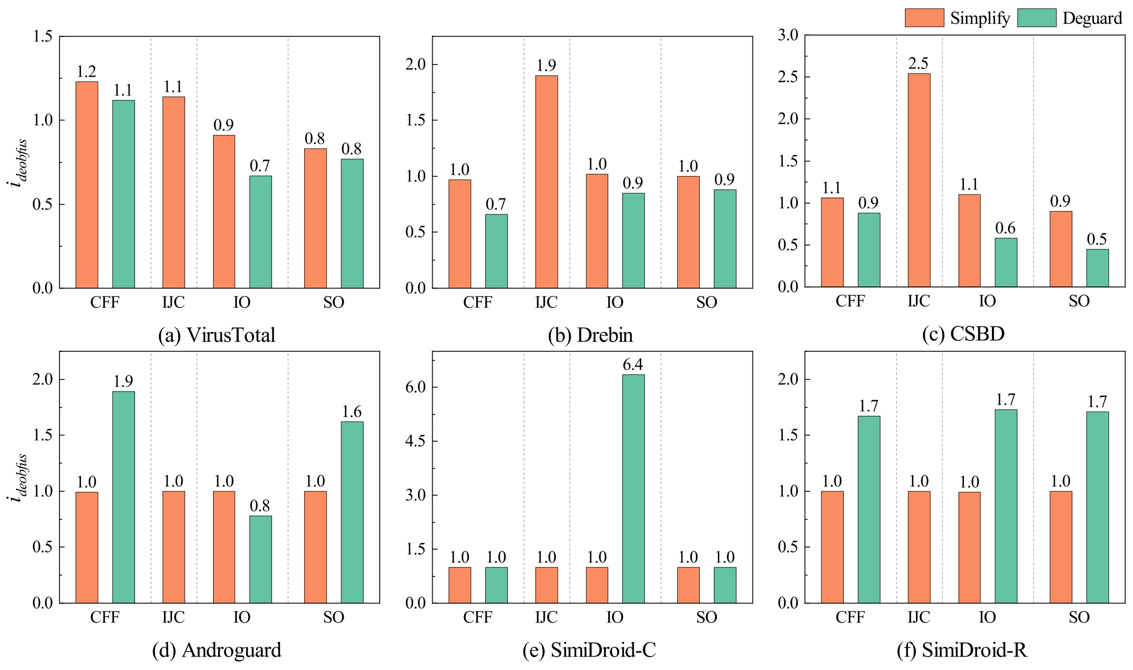


Figure 4.4: The impact of code deobfuscation tools on different detectors. (a) VirusTotal, (b) Drebin, (c) CSBD, (d) Androguard, (e) SimiDroid-C, and (f) SimiDroid-R.

By analyzing the experimental results, after deobfuscation, it can be found that many detectors' precision decreased for obfuscated malware under different strategies. The reason is that, in many scenarios, the deobfuscators can not reinstate the obfuscated code back to the original code but adding additional 'noise' (the modified code or the newly introduced code fragments by the deobfuscators); therefore, the results of the malware detectors can be negatively impacted.

4.3.3 RQ3. Impact of Deobfuscation Tools

To compare the impact of deobfuscation on the detection precision using different deobfuscation tools (i.e., Simplify and Deguard), we defined a parameter $i_{deobfus}$ as follows.

$$i_{deobfus} = \frac{acc_{deobfus}}{acc_{obfus}} \tag{4.2}$$

where $acc_{deobfus}$ denotes the precision of detecting deobfuscated piggybacked apps, as shown in Tables 4.5 and 4.6, while acc_{obfus} is the precision of detecting obfuscated piggybacked apps, as depicted in Table 4.4. Note that If $i_{deobfus}$ is larger than 1,

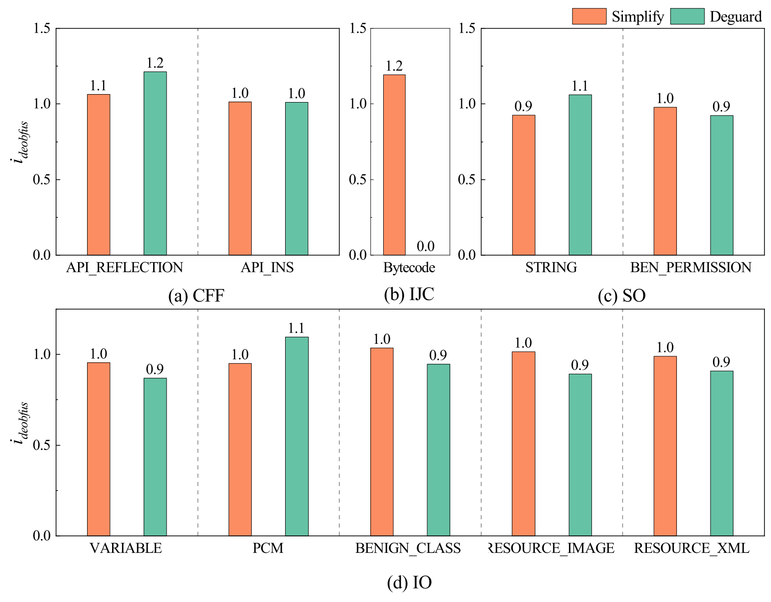


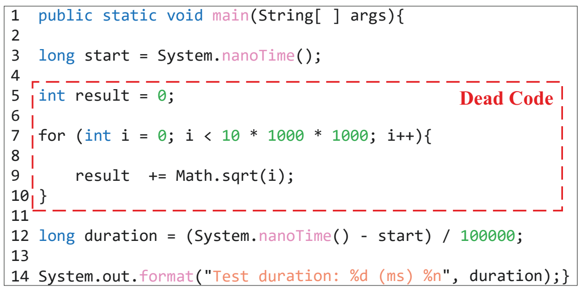
Figure 4.5: The impact of code deobfuscation tools on different types of obfuscation strategies. (a) CFF, (b) IJC, (c) SO, and (d) IO.

it means the detection precision has increased after deobfuscation. Otherwise, it means the detection precision has decreased after deobfuscation.

Impact on Detectors. Figure 4.4 shows the impact of code deobfuscation tools on different detectors. For each detector, we compare the impact of each strategy type using different code deobfuscation tools. Note that each column in the figure is calculated by averaging the $i_{deobfus}$ belonging to the corresponding type of strategy and detector.

We can observe from Figure 4.4(a), (b), and (c) that Simplify has a more significant impact on commercial anti-malware products (i.e., VirusTotal) and machine learning-based detectors (i.e., Drebin and CSBD) than Deguard. For example, after Simplify deobfuscation with the strategy types of CFF and IO, the detection precision of CSBD has dramatically improved, compared with the deobfuscation using Deguard.

Deguard has a more significant impact on similarity-based detectors (i.e., Androguard and SimiDroid) than Simplify, as shown in Figure 4.4(d), (e) and (f). For example, after deobfuscation using Deguard under the strategy types of CFF, IO, and SO, the detection precision of SimiDroid-R has dramatically improved, compared with the deobfuscation using Simplify.



```
public static void main(String[ ] args){

long start = System.nanoTime();

int result = 0;

for (int i = 0; i < 10 * 1000 * 1000; i++){

    result  += Math.sqrt(i);
}

long duration = (System.nanoTime() - start) / 100000;

System.out.format("Test duration: %d (ms) %n", duration);}
```

Figure 4.6: Case study: insert junkcode.

Finding #7: *Different code deobfuscation tools have a different impact on anti-malware detectors. In particular, by using Simplify for commercial anti-malware products (i.e., VirusTotal) and machine learning-based detectors (i.e., Drebin and CSBD) the improvement in the precision of detecting deobfuscated piggy-backed apps is greater. However, for similarity-based detectors (i.e., Androguard and Simidroid), Deguard performs better than Simplify.*

Impact on Obfuscation Strategies. Figure 4.5 shows the impact of code deobfuscation tools on different types of strategies. Note that each column in the figure is calculated by averaging the $i_{deobfus}$ belonging to the same strategy and corresponding to the same type. From Figure 4.5, we can not conclude that the code deobfuscation tools have any significantly different impact on each strategy in the same type when detecting deobfuscated piggybacked apps after code deobfuscation.

4.4 Case Studies

This section presents case studies to demonstrate representative deobfuscated piggybacked apps using different deobfuscation tools.

Insert Junk Code (IJC). Malware ID-FFCE7D9B[1] is deobfuscated by Simplify using an insert junk code (IJC) obfuscation strategy. This strategy involves obfuscators inserting extra code snippets into the script, including some variables

[1]Since each malware has a unique hash value (MD5), we used the first eight digits to denote its hash values.

```
private void CalculatePayroll(SpecialList a){
    while (a.HasMore()){
        b = a.GetNext(true);
        b.UpdateSalary();
        DistributeCheck(b);
    }
}
```

(a) Obfuscated Code

```
private void CalculatePayroll(SpecialList DBhelper){
    while (DBhelper.HasMore()){
        DB = DBhelper.GetNext(true);
        DB.UpdateSalary();
        DistributeCheck(DB);
    }
}
```

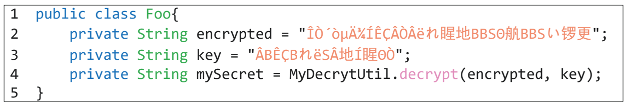
(b) Deobfuscated Code

Figure 4.7: Case study: identifier renaming.

and functions that are never referenced or called after they are fixed, as shown in Figure 4.6. This code may be executed, but it will not affect the overall execution result of the script. The ideal result would be that deobfuscators remove it from the code. However, Simplify cannot discern and delete the IJC. Therefore, the code structure of this obfuscated malware can not be restored to the original app. Consequently, the deobfuscation does not contribute to more accurate detection of this malware. This malware eventually evades both machine learning and similarity-based detectors. VirusTotal's detection rate has decreased compared to the obfuscated one. The newly generated junk code introduces additional noise that adversely affects the anti-virus vendors' performance.

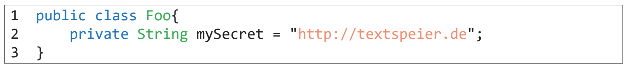
Identifier Obfuscation (IO). Malware ID-E4A8A133 is deobfuscated by Simplify for identifier obfuscation (IO), which is the most common obfuscation technique. For example, a class/method/variable name is often obfuscated into an arbitrary identifier composed of uppercase and lowercase letters and numbers, which can be used to evade machine-learning-based detectors that rely on recognizing and matching identifiers. However, the similarity-based detector can detect this malware. Although Simplify can replace some obfuscated with more meaningful names (e.g., an example is illustrated in Figure 4.7), the deobfuscation is still unable to restore the app to the original malware, thus also evading the machine-learning-based detectors.

String Obfuscation (SO). Malware ID-A9D69887 is deobfuscated by Deguard against string obfuscation. Figure 4.8 indicates that the decryption class



```java
public class Foo{
    private String encrypted = "ÎÒ´òµÄ¾ÍÊÇÂÒÂëれ腥地BBSΘ航BBSい锣更";
    private String key = "ÂBÊÇBれëSÂ地Í腥ΘÒ";
    private String mySecret = MyDecrytUtil.decrypt(encrypted, key);
}
```

(a) Obfuscated Code



```java
public class Foo{
    private String mySecret = "http://textspeier.de";
}
```

(b) Deobfuscated Code

Figure 4.8: Case study: string obfuscation.

generates a key through the hash code value of the APK certificate and the internal key reference array (ref_key) and reads and decrypts the original encrypted DEX file. Unlike using AES for the string encryption option, this class encryption option uses its decryption algorithm. For this malware, after deobfuscation, the string is decrypted to be a fraudulent e-commerce website. Thus, many more vendors in VirtualTotal can detect this malware after deobfuscation.

4.5 Discussions and Limitations

In this chapter, we chose VirusTotal for testing commercial anti-malware products, however, we have two concerns. First, VirusTotal integrates dozens of antivirus products on their website. However, whether the website version and desktop version of same antivirus product will get exactly the same detection results is unclear, which may lead to bias in the observe results. Second, some antivirus products will be replaced or deleted on VirusTotal. Also, some antivirus products will also be upgraded to new versions on VirusTotal, resulting in changes of the detection ability against same Android malware, which will also lead to changes in the observed results over time.

CHAPTER 5

Active Learning Based Ranking for Adversarial Android Malware

5.1 Introduction

Security issues pertaining to Android phones are becoming increasingly serious. Therefore, Android malware detection has received increasing attention recently [1, 6, 21, 23–28, 30–33]. In the battle between malware and antivirus software, malware evading techniques have been gradually improved. Especially among all Android malware, adversarial attacks are very popular to use [40–46, 115]. Through penetration testing, industry companies and mobile users can determine whether the system has hidden vulnerabilities and security risks from an attack point of view. The testing conducted through generating and exercising adversarial samples against the antivirus engines of a software system can also gain a deeper perception of its security and robustness.

Code obfuscation is one of the most commonly used methods to generate adversarial Android malware [76–78]. Different obfuscation strategies will have a different effect on evading antivirus products. For example, Figure 5.4 shows the scoring of two obfuscated samples of an original Android malware. It is an example show-

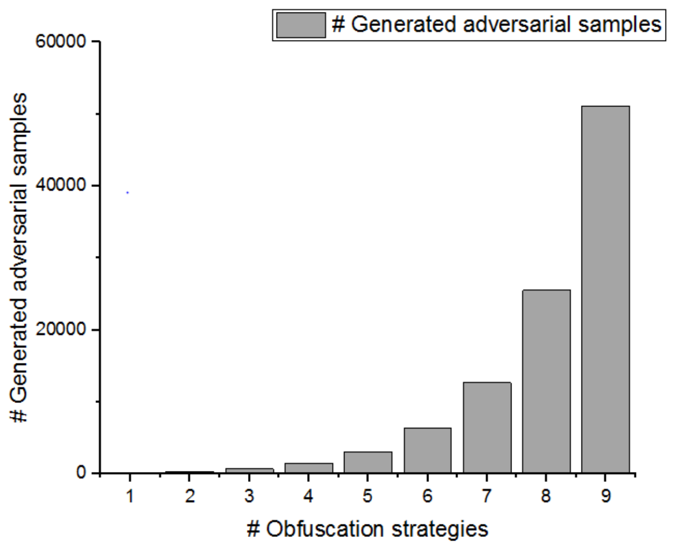


Figure 5.1: Number of generating 100 malware's adversarial samples given # of obfuscation options.

ing the scores of the adversarial samples of an original Android malware, which has been obfuscated by *Package renaming* + *Permission encryption* and *String encryption* respectively. These samples are then uploaded to VirusTotal, and they obtain different ranking scores from VirusTotal.

Note that if more obfuscation strategies are involved, the scale of adversarial samples will be increased exponentially. Many available strategies can be utilized to obfuscate Android malware, which can be further combined to generate adversarial samples. For k obfuscation strategies, $2^k - 1$ adversarial samples can be generated using only one Android malware as shown in Figure 5.1. For example, combining the nine obfuscation options can generate 511 adversarial samples using one single malware. The open-source Android application obfuscation tool - Obfuscapk [116] offers more than 16 single obfuscation strategies. If we combine the 16 obfuscation options, 65,535 adversarial samples can be generated using one single malware.

Therefore, it is impossible to utilize all generated adversarial samples for penetration testing. The executable strategy is to pick up the samples with the most potent evading capability to save penetration testing costs.

Hence, it is essential to develop a method to select adversarial samples that have the most potent ability to evade antivirus products at a much lower cost. Ideally, the top evasive samples should be identified accurately before the testing, which

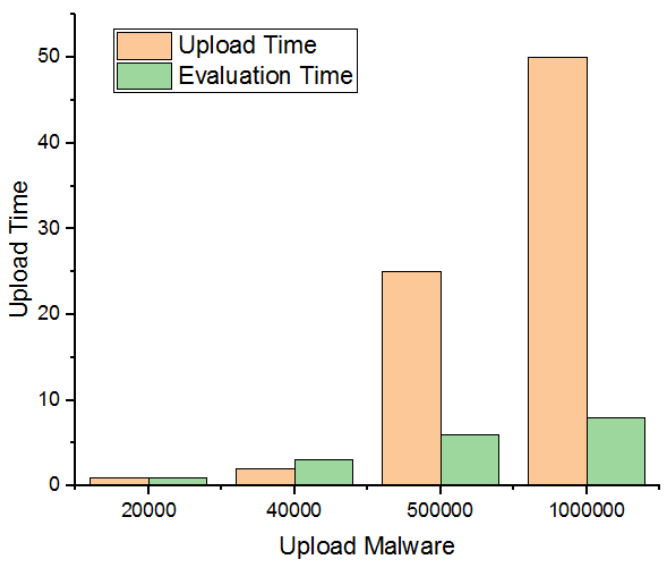


Figure 5.2: Time cost to process apps by VirusTotal.

requires the approach to achieve comparable precision in ranking the top evasive samples like that of state-of-the-practice antivirus engines.

We propose an active learning approach, ALR, to precisely rank the adversarial samples to solve the above challenge. First, we extract features from the bytecode and manifest file. Each Android application contain a manifest file, which supporting app installation and subsequest execution on the smartphone. Also, we desassemble the bytecode to access necessary information such as API calls, etc [22]. Second, we build the control flow graph (CFG) of adversarial samples, as CFG is a useful abstraction to model the execution order. Then, we use the active learning method, which automatically selects part of the data requesting labels from the data set using automatic machine learning algorithms. It actively designs a proper query function, continuously picks out data from the unlabeled data, and adds it to the training set after adding the label. The proposed active learning can precisely produce a small but representative training set. Subsequently, we adapt the two views to extract features and train 6 LTR models under each view (i.e., 12 ranking models in total) using the 6 popular LTR models, MART [117], RankBoost [118], AdaRank [119], Coordinate Ascent [120], LambdaMart [121], and Random Forest [122]. The novelty lies in the MVML active learning method's successful application to precisely select and ranks the most evasive samples, thus supporting efficient adversarial sample

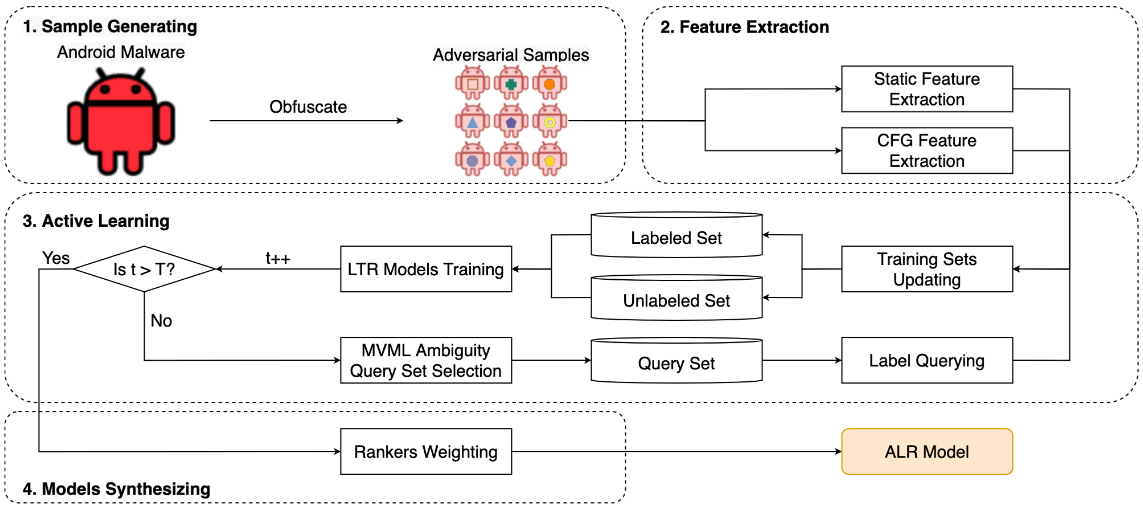


Figure 5.3: The overview of ALR.

testing to avoid the high cost of antivirus engines like VirusTotal.

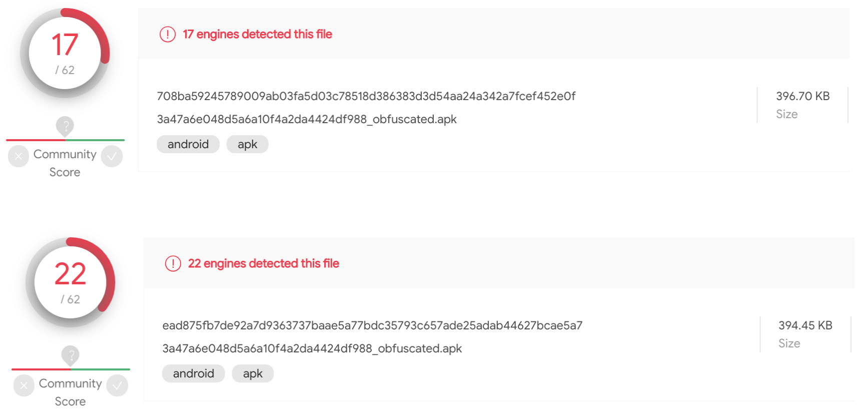
5.2 Our Approach

Figure 5.3 shows the overview of ALR. ALR comprises four stages: 1. Adversarial Sample Generation. 2. Feature Extraction. 3. Active Learning. 4. Ranking Models Synthesizing. As a part of our approach, ALR will automatically sample some adversarial samples and upload them to VirusTotal to get the ranking scores. This section details the key components of active learning and model synthesizing.

5.2.1 MVML Active Learning

Active learning uses automatic machine learning algorithms to automatically select some data request tags from the data set, which is also known as query learning or optimal experimental design in statistics. After designing a proper query function, active learning continuously selects data from unlabeled data and adds it to the training set.

When using traditional supervised learning methods for classification, the larger the training sample size is, the better the classification result will be. However, it is challenging to obtain labeled samples in many real-world scenarios, which require experts to label manually, incurring the high time and computational costs. There-



17 / 62

17 engines detected this file

708ba59245789009ab03fa5d03c78518d386383d3d54aa24a342a7fcef452e0f
3a47a6e048d5a6a10f4a2da4424df988_obfuscated.apk

android apk

396.70 KB
Size

Community Score

22 / 62

22 engines detected this file

ead875fb7de92a7d9363737baae5a77bdc35793c657ade25adab44627bcae5a7
3a47a6e048d5a6a10f4a2da4424df988_obfuscated.apk

android apk

394.45 KB
Size

Community Score

Figure 5.4: An example of adversarial samples on VirusTotal.

fore, it is necessary to find a way to use fewer training samples to obtain a better classifier. Active learning queries the most useful unlabeled samples using specific algorithms, passes them to experts for labeling, and then uses the queried samples to train the classification model to improve its accuracy.

Active learning methods can be roughly categorized into multi-view or single-view, multi-learner, or single-learner. There are four combinations of active learning methods: single-view single-learner (SVSL), multi-view single-learner (MVSL), single-view multi-learner (SVML), and multi-view multi-learner (MVML). SVSL use one source of view, and requires learners to avoid large induction bias. Conversely, MVSL actively learns to infer queries using multiple sources of views, but MVSL can be further improved through integration. SVML leverage the multiple learners' result on one training set. However, if multiple views can be applied, there is still room for improvement. Therefore, to conquer the limitations, MVML method is proposed [123].

MVML uses a similar process in co-testing [124] to select malware to be labeled. First, there is a small number of labeled adversarial malware samples in set L and many unlabeled samples in set U. The labeled examples train several LTR models with multiple views in set L. In each iteration, unlabeled examples in set U are predicted by the trained models. Then, each LTR model estimates the samples in each view and produces scores. Those ambiguous samples are selected as a query set Q. Then, the ground truth labels of the query set will be obtained, and the newly labeled samples will be moved to L. Subsequently, each LTR model is trained with refreshing labeled set L, and repeat the whole process several times. At the classification phase, we calculate this by combining the results of the LTR models of these views to determine the final hypothesis.

To measure the within-view disagreement among all LTR models in the i^{th} view V_i, we use qualified entropy to select samples. The entropy includes M views $V_1, ..., V_m$ with K LTR models and $n^i_j(x)$, the number of LTR models in the same score j considering the sample x in the learner l:

$$j = score(x, i, l) \tag{5.1}$$

$$P^i_j(x) = \frac{n^i_j(x) + \frac{\varepsilon}{2}}{K + \varepsilon} \tag{5.2}$$

where ε is a small positive constant ensuring nonzero confidence, and n^i_j denotes to the number of models, regarding to the sample x, with score j within view V_i. We use P^i_j to represent $P^i_j(x)$ for short. Then the $entropy^i$ within view V_i can be deduced by P^i_j as:

$$entropy^i(x) = -\sum_{j=0}^{T} P^i_j log_2(P^i_j) \tag{5.3}$$

where T is the maximum score of VirusTotal.

To measure the LTR models' uncertainty between different views, we introduce the ambiguity of a sample x as:

$$ambiguity^{a,b}(x) = \left\{ \begin{matrix} 1 + P^a_{j_a} log_2(P^a_{j_a}) + P^b_{j_b} log_2(P^b_{j_b}), O_a \neq O_b \\ -1 - P^a_{j_a} log_2(P^a_{j_a}) - P^b_{j_b} log_2(P^b_{j_b}), O_a = O_a \end{matrix} \right\} \tag{5.4}$$

where O_a and O_b are output arrays of the ranking scores made by two views, V_a and V_b, and P^i_j is the confidence, indicating that the $a-th$ view V_a considers the predicted sample x as learner j, where j is the learner maximising P^i_j among learners, that is:

$$j_a = \underset{j}{argmax}(P^a_j), j_b = \underset{j}{argmax}(P^b_j) \tag{5.5}$$

Table 5.1: Obfuscation Strategies

Type	Strategy	Detailed Description
CFF	API reflection	Hides all APIs by Java reflections.
	API injection	Inserts invalid APIs between existing APIs.
IO	Variable encryption	Renames the variable name in the source code, then replaces it with a meaningless identifier, making it harder to crack this analysis.
	Package renaming	Obfuscate the package name in the same way as the variable name.
	Class renaming	Obfuscate the class name in the same way as the variable name.
	Image encryption	Modifies directly at the source level, then replaces the code and renames "icon.png" to "a.png", lastly hands it over to Android for compilation.
	XML encryption	Modifies directly at the source level, then replaces the code and "R.string.name" in xml file with "R.string.a", and hands it over to Android for compilation.
SO	String encryption	Encrypts the string locally, then hardcodes the ciphertext into it. Lastly, decrypts it at runtime.
	Permission encryption	Encrypts the permission locally, hardcodes the ciphertext into it, and decrypts it at runtime.

* The abstract in Type: SO-String Obfuscation, IO-Identifier Obfuscation, IJC-Insert Junk Code, CFF-Control Flow Flattening.

The MVML methods combine two methods and define the ambiguity as:

$$ambiguity'(x) = \sum C_M^2 ambiguity + \sum_{i=1}^{M} entropy^i \tag{5.6}$$

where ambiguity is the disagreement between multiple views, C_M^2 denotes a combination of two views among M views. $entropy^i$ is the within-view disagreement between multiple LTR models of V_i.

5.2.2 LTR Models Synthesizing

In this section, we introduce the six chosen LTR models in detail, namely Multiple Additive Regression Trees (MART), RankBoost, Adarank, Coordinate Ascent, LambdaMart and Random Forests.

Multiple Additive Regression Trees (MART). MART is a gradient-boosting decision tree (GBDT) algorithm. GBDT's core idea is that in a continuous iteration, the regression decision tree model generated by the new round of iterations fits the gradient of the loss function, and finally, all the regression decision trees are superimposed to obtain the final model.

RankBoost. The idea of RankBoost is the conventional idea of binary learning to rank. By constructing the target classifier, the objects in the pair have a relative size relationship. In layman's terms, this involves grouping objects into a pair of pairs, such as a set of $r1 > r2 > r3 > r4$, that can constitute a pair: ($r1$, $r2$) ($r1$, $r3$), ($r1$, $r4$), ($r2$, $r3$) ($r3$, $r4$). These pairs have a positive value, that is, the label is 1; and the remaining pairs such as ($r2$, $r1$) have a value of -1 or 0. This ranking task is transformed into a classification problem.

Adarank. Xu and Li proposed the boosting-based learning algorithm AdaRank [119] to optimize all kinds of information retrieval evaluation criteria directly. AdaRank is a direct application of the Adaboost [125] framework. It directly integrates the evaluation metrics and the adjustment process of the model's weights for ensemble learning. AdaRank constructs an exponential loss function based on information retrieval evaluation criteria. Under the assumption of the additive model, a stepwise optimization method is used to optimize the loss function.

Coordinate Ascent. Coordinate Ascent is a non-gradient optimization method. It achieves the purpose of optimizing the function by updating one dimension of the multivariate function each time and multiple iterations until convergence. In short, it constantly selects a variable for one-dimensional optimization until the function reaches the local best. Coordinate Ascent is suitable for solving large-scale and complex problems with smooth objective functions.

LambdaMart. LambdaMart is a listwise type LTR algorithm based on the LambdaRank algorithm and the MART (Multiple Additive Regression Tree) algorithms and transforms the search engine result ranking problem into a regression decision tree problem. LambdaMart uses a particular lambda value to replace the above gradient, adding the LambdaRank algorithm and the MART algorithm. As a gradient boosting tree algorithm, LambdaMART allows missing values and does not require data standardization.

Random Forests. The construction process of random forests: 1) use the bootstrapping method to randomly sample m samples from the original training set and take n_tree samples in total to generate n_tree training sets; 2) for n_tree training sets, we train n_tree decision tree models separately; 3) for a single decision tree model, assuming that the number of features of the training sample is n, then each split is based on the information gain ratio to selects the best feature to split; 4) each tree is to split in a way by knowing that all the examples of this node belong to the same category. No pruning is required during the splitting of the decision tree; 5) a random forest is composed of the multiple decision trees generated.

As ALR is trained according to two views, namely static and CFG features, each view can train six ranking models, which means we train 12 ranking models, respectively. In the process of selecting the sample with the strongest evading ability, if 12 ranking models are considered in a balanced manner, too many irrelevant adversarial samples will be covered. Although this will not affect recall, it will reduce the precision of the sample with the strongest evading ability. Therefore, we decide to give the selection range and assign weights to different sub-models to improve the precision score. Under this premise, a straightforward method is used to give a reference weight w according to the accuracy of the sub-model. The more accurate the sub-model, the greater its weight and the stronger the influence on the final prediction:

$$w_i = \frac{NDCG(C_i)}{\sum_1^{12} NDCG(C_i)} \tag{5.7}$$

where C_i represents each ranking model and w_i represents the weight of each ranking model. NDCG is short for normalized discounted cumulative gain, which is introduced in Equation 5.12 in detail. Table 5.7 shows the weights according to the NDCG score on the training set of each LTR model after training. A detail description is given in the Evaluation section.

After a given total number of candidates, we have to decide how many candidates should be selected for each ranking model.

$$n_{C_i} = w_i * N^k \tag{5.8}$$

where n_{C_i} is the final ranking score of candidates that considering the weights of all ranking models C_i. N^k is the ranking score of candidates given by the corresponding ranking models.

Table 5.2: Adversarial samples for evaluation

	Original Malware	Adversarial Samples
Training Set	300	56,146
Testing Set I	100	21,512
Testing Set II	100	22,342
Total Samples	500	100,000

5.3 Data Collection and Aggregation

5.3.1 Adversarial Sample Generation

To generate adversarial samples, ALR uses each of the nine obfuscation strategies in AVPASS [126] and their combinations. AVPASS can use obfuscation technology to construct an adversarial APK to bypass antivirus detection. It can automatically deform the APK so that an antivirus will mistake a malicious app for a benign app. We automatically use AVPASS's 9 APK (Android installation package) obfuscation modules to generate various obfuscated APKs. By applying different obfuscation methods and their combinations to an original malware app (virus APK file), many semantically equivalent variants of that malware can be generated. In this chapter, the nine strategies of AVPASS are categorized into three types, i.e., control flow flattening, identifier obfuscation, and string obfuscation, as shown in Table 5.1.

5.3.2 Static and CFG Feature Extraction

First, we collect as many features as possible from the application for static analysis. The collection of features imitates Drebin [22], which includes four sets of features from the manifest file (i.e., permissions, hardware components, app components, and filtered intents) and four sets of features from the disassembled code (i.e., restricted API calls, user permissions, suspicious API calls, and network addresses).

These functions are organized in string sets and embedded in the joint vector space, for example, the vector space associated with corresponding permissions, intentions, and API calls that send advanced SMS messages to specific areas.

Additionally, we extract another type of feature from an app's control flow graphs (CFGs), which represents the execution order of an application, i.e., the possible flow of execution of all basic blocks in a process in the form of a graph. It is an abstract data structure often used during compile-time and maintained internally by the compiler.

Static Features. We first collect as many features as possible from the adversarial samples from static analysis. Eight sets of features are collected. We use the Android Asset Packaging Tool (AAPT) [95] to efficiently retrieve the information stored in the manifest file. Four feature sets are extracted from the manifest file: hardware components, requested permissions, application components, and filtered intents. The other four features are extracted from the disassembly code: restricted API calls, used permissions, suspicious API calls, and network addresses.

Then these features are organized into a collection of character strings, embedded into a vector space, and the patterns and combinations of these features are analyzed geometrically. Finally, using machine learning, the embedded feature set identifies Android system malware and interprets the detection results. The extracted feature sets include:

- **S1 Hardware Components.** It is a security risk to request access to specific hardware, as the use of specific hardware combinations may reflect harmful behavior. For example, malicious applications that can access GPS and network modules may leak location data to attackers via the network.

- **S2 Permission.** In Android OS, the permission mechanism is one of the principal security mechanisms,which is needed to be granted by users for installation and accessing security-related resources.

- **S3 App Components.** There are four application components, each component defining a different interface within the system, i.e., broadcast receivers, content providers, services, and activities. They are collected as they can help identify malware' common used components.

- **S4 Filtered Intent.** We choose intents as one of our feature set because Android malware usually follow some specific intents. For example, malware usually use the intent - `BOOT COMPLETED` to trigger malicious activity immediately after restarting the smartphone.

- **S5 Restricted API Calls.** A special case that reveals malicious behavior is a restricted API call, which indicates that the hackers leverage root vulnerabilities to exceed the limits of Android OS.

- **S6 Used Permissions.** S5 extracted the complete call set to determine the requested and used permission subset. Also, one permission can protect several API calls.

- **S7 Suspicious API Calls.** Malware samples leverage certain API calls broadly as they can access user's sensitive data. As some particular hazardous behavior are trigered by certain suspicious API calls, we extract them as one of feature set.

- **S8 Network Address.** The malware periodically establishes a network connection to receive commands from the hackers or send out sensitive data. Therefore, network address is chosen as the last feature set.

Control Flow Graph (CFG). We also construct an Android application's CFG. We collect all the fundamental blocks that makeup and call it the function of the application. A basic block represents a series of CFG's instruction, including only one entry point and one exit point. A basic block is the smallest part of an application which is executed.

To extract the abstraction of CFG, we use a method based on the syntax proposed by Cesare et al. [127] to transform the extracted CFG as a string, which is an abstraction of the programs' code. It contains the structure information of the Android app's source code, however, drop the fine grained information.

First, we define the equation:

$$BB_{all} = \{BB_1, BB_2, ..., BB_n\} \tag{5.9}$$

where BB_i represents a basic code block. BB_{all} represents the set of the n source code blocks come upon.

Regarding each App and their adversarial samples, we define a list $Features_{App}$, of binary values. If the value is 1, it means the basic source code blocks from BB_{all} appear in the program and 0 represent those blocks that do not appear.

$$Features_{App} = (b_{App,1}, b_{App,2}, ..., b_{App,n}) \tag{5.10}$$

if the basic block BB_i is is encountered, b_{App}, i is set to 1, and b_{App}, i is set to 1 otherwise.

5.4 Evaluation

The objective of our evaluation is to show that our method is effective in ranking Android malware adversarial samples.

5.4.1 Experimental Setup and Implementation

To evaluate the effectiveness of ALR, we generate 100,000 adversarial samples, which are generated from 500 malware that were manually collected from Jan 2020 to Mar 2020 on VirusShare [94]. Table 5.2 shows how the generated adversarial samples are divided into different sets for evaluation. In all the adversarial samples generated by all 500 malware, we must first classify them into the training and test sets. First, we select all the adversarial samples generated by 100 of the 500 malware and put them into the test set I, which are picked randomly. These adversarial samples will be used to evaluate the effectiveness of ALR and calculate the weights of each LTR model. Then we select all the adversarial samples generated by 100 of the 400 malware and put them into the test set II. The adversarial samples in this set will be used to calculate the recall result of covering top-ranked adversarial samples. In the end, 21,512 adversarial samples are selected into the testing set I, and 22,342 adversarial samples are in the test set II. All the remaining 56,146 adversarial samples are selected into the training set.

We use AVPASS to generate adversarial samples for each Android malware using nine obfuscation strategies and their combination strategies. Ideally, 500 malware can have 255,500 variants based on the nine strategies and their combinations. However, some strategy combinations failed to yield a correct or executable app by AVPASS; therefore, we only successfully generated 100,000 adversarial samples. In theory, each original malware will generate 512 adversarial samples. However, not every original malware can generate 512 adversarial malware. Commonly, some strategies will fail when generating adversarial samples. It is worth noting that the failed obfuscation strategies regarding different original malware are different. The possible reason is that different original malware uses different libraries or uses different setups.

Regarding the training process, we use the MVML active learning method to train our model; with each round of training, we select 500 samples into the query set, and a total of 40 rounds of training were conducted. Finally, we selected 20,000 samples from the 56,146 samples. The ranking scores as ground truth are obtained by uploading samples to VirusTotal.

We evaluate the effectiveness of our approach from the following aspects: 1) we evaluate the performance of applying the MVML active learning method to collect the training set; 2) we discuss which obfuscation and combination strategies are more likely to produce strong evasive adversarial samples.

5.4.2 Evaluation Metric

In order to evaluate which ranking model has better ranking performance, we use normalized discounted cumulative gain (NDCG) as the evaluation metric. MRR takes the standard ranking results' reciprocal as its accuracy and then averages all the ranking tasks. In MAP, the samples and the ranking query are either related or

Table 5.3: NDCG score of using Random Static method

LTR model \ Feature Dimension	K=50	K=100	K=150	K=200
MART	0.225	0.243	0.435	**0.549**
RankBoost	0.220	0.232	0.247	**0.414**
AdaRank	0.125	0.311	0.497	**0.655**
Coordinate Ascent	0.109	0.134	0.201	**0.628**
LambdaMart	0.223	0.346	0.419	**0.690**
Random Forest	0.356	0.385	0.444	**0.612**

Table 5.4: NDCG score of using MVML Static method

LTR model \ Feature Dimension	K=50	K=100	K=150	K=200
MART	0.766	0.774	0.917	**0.924**
RankBoost	0.714	0.749	0.923	**0.977**
AdaRank	0.823	0.849	0.931	**0.974**
Coordinate Ascent	0.633	0.682	0.952	**0.985**
LambdaMart	0.655	0.657	0.732	**0.993**
Random Forest	0.592	0.604	0.715	**0.846**

unrelated. That is, the relevance is not 0 or 1. NDCG is improved as the correlation is divided into $r + 1$ levels from 0 to r (r can be set); therefore, we chose NDCG to evaluate the ranking algorithms.

The premise of DCG is that high-relevance adversarial samples appearing in lower-ranking scores should be punished, and the graded relevance value decreases logarithmically with the position of the results. The traditional DCG accumulation formula is defined as:

$$DCG_{C_i} = \sum_{i=1}^{C_i} \frac{rel_i}{log_2(i+1)} = rel_1 + \sum_{i=2}^{C_i} \frac{rel_i}{log_2(i+1)} \tag{5.11}$$

where C_i is the total number of the used ranking model and rel is the rank score of the candidate i, which is manually scored data. Then, the ideal discounted cumulative gain (IDCG) is calculated, that is, the perfect sequence DCG; the calculation method is also the same as step 1, except that the sequence is not derived by the algorithm, but is the best sequence that is manually discharged from the sequence according to certain evaluation criteria. According to the results of the previous two

Table 5.5: NDCG score of using Random CFG method

LTR model \ Feature Dimension	K=50	K=100	K=150	K=200
MART	0.165	0.270	0.640	**0.682**
RankBoost	0.19	0.258	0.261	**0.406**
AdaRank	0.154	0.203	0.300	**0.399**
Coordinate Ascent	0.146	0.286	0.383	**0.478**
LambdaMart	0.182	0.218	0.469	**0.737**
Random Forest	0.150	0.217	0.234	**0.402**

Table 5.6: NDCG score of using MVML CFG method

LTR model \ Feature Dimension	K=50	K=100	K=150	K=200
MART	0.711	0.936	0.944	**0.944**
RankBoost	0.496	0.827	0.857	**0.895**
AdaRank	0.652	0.712	0.738	**0.818**
Coordinate Ascent	0.561	0.780	0.807	**0.845**
LambdaMart	0.492	0.774	0.8737	**0.891**
Random Forest	0.709	0.733	0.839	**0.929**

steps, the calculation formula of NDCG is as follows:

$$NDCG_{C_i} = \frac{DCG_{C_i}}{IDCG_{C_i}} \tag{5.12}$$

In this experiment, we use the recall of $Top\ k$ as the evaluation metric, which represents the ratio of the correct predicted samples to all the samples in the $Top\ k$ ranking score list. For example, Top 10 means selecting the Top 10 adversarial

Table 5.7: Weights of each learner

Learner	MVML static	MVML CFG
MART	0.074	0.094
RankBoost	0.069	0.087
AdaRank	0.095	0.074
Coordinate Ascent	0.096	0.076
Lambdamart	0.097	0.086
Random Forest	0.069	0.083

samples in the ranking score list.

$$R^k = \frac{\sum T^k_{C_i}}{G^k} \tag{5.13}$$

where R^k is the recall of $Top\ k$, $\sum T^k_{C_i}$ represents the true positive samples which are selected by ALR and G^k represents the samples in the $Top\ k$ ranking score list.

5.4.3 Result Analysis

This section presents our evaluation analysis regarding the usefulness of applying the active learning method and the effectiveness of the obfuscation strategy and their combinations.

Evaluating effectiveness and efficiency of our MVML active learning approach. We compare the MVML active learning and random sampling method to show the active learning method's applicability and effectiveness in improving LTR models' performance. Tables 5.3, 5.4, 5.5 and 5.6 demonstrate the comparison of NDCG scores that randomly select adversarial samples and the MVML active learning method to select the training set. These tables show that our approach performs better than randomly selecting the training set method. **Random Static & Random CFG** mean adversarial samples are randomly selected into the training set. **MVML Static & MVML CFG** mean using the MVML active learning method to select the most informative samples into the training set.

In general, the MVML active learning method achieves a higher score than the random methods' results. Given the learned representation, we compare the ranking performance of MART, RankBoost, AdaRank, Coordinate Ascent, Lambdamart, and Random Forest. To compare the different LTR models' results fairly, we keep their feature representation embeddings unchanged, the same representation space with the dimensions K = 50, K=100, K=150, K=200. Considering the results of the four tables, if the dimension K = 200, the LTR models can have better NDCG scores than other representation spaces. In Table 5.3 and 5.5, the random sampling method achieves an NDCG score of 0.690 out of 1 with the static feature representation using the Lambdamart and 0.737 with CFG feature representation.

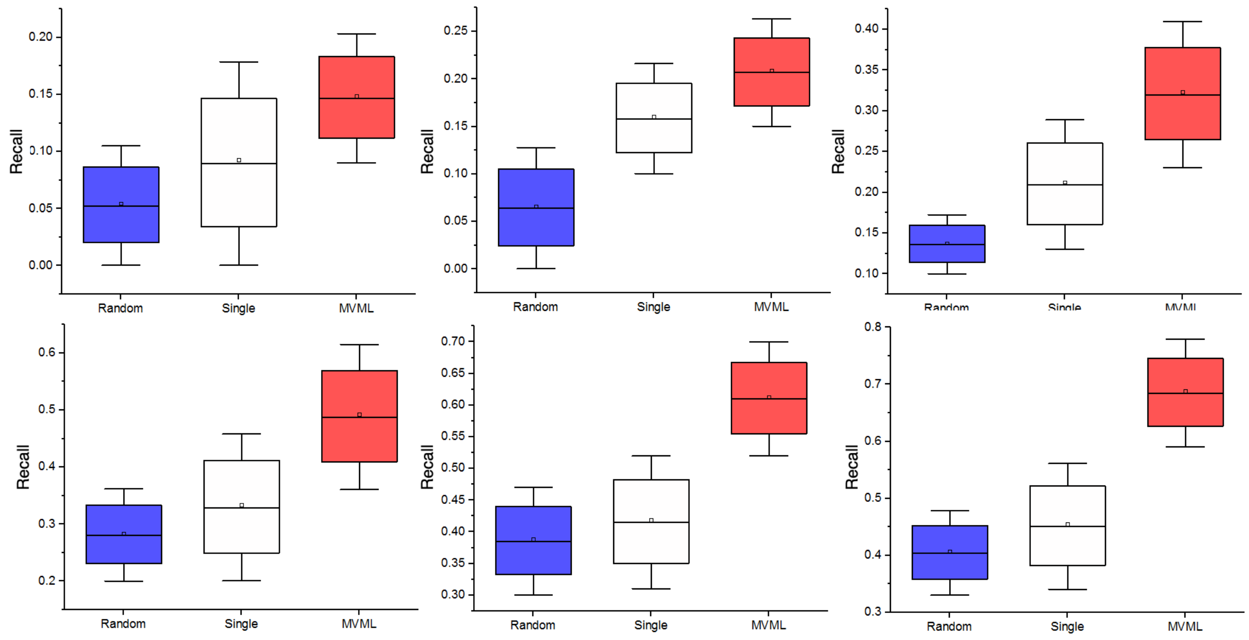


Figure 5.5: Recall results of ranking adversarial samples.

Nonetheless, in Tables 5.4 and 5.6, MVML active learning achieves an NDCG score of 0.993 using static features with the Lambdamart model and an NDCG score of 0.944 using the CFG feature with the MART models. Furthermore, based on the obtained NDCG scores, the well-trained LambdaMart model using the MVML static method is the best single LTR model, which will be used to evaluate if our LTR models synthesizing can achieve better performance.

Verifying that the model recognizes the effect of the models top adversarial samples' coverage. This section compares the MVML active learning and random sampling method to show that MVML active learning methods are better at covering the top-ranked adversarial samples. In order to pick up the top-ranked adversarial samples, we obtain the VirusTotal score as the standard ranking score to evaluate the LTR models' scoring performance. Subsequently, we categorize the adversarial samples into the Top 10, and the Top 20, Top 30,...etc. For each range of $Top\ k$, we specify the quantity of selected samples and select the corresponding amount of samples according to the weight of each of the 12 ranking models. At the same time, we verify how many of the samples scored by these rankers are also in the corresponding range of standing ranking scores.

Figure 5.5 demonstrates the recall result of covering the top-ranked adversarial samples. The Y-axis of Figure 5.5 indicates the recall of Top 10, Top 20...Top 100. For instance, 0.21 in Top 10 means 0.21 *10=2.1, that is, of all the samples rated Top 10 by ALR, 2 samples are in the ranking score list Top 10. In addition, for Top 10, Top 20, Top 30, Top 40 and Top 50, etc., we stipulate that the maximum number of samples that can be selected are 10, 20, 30, 40 and 50, etc., that is, the number of selected candidates should be the same as k. Additionally, as the precision of $Top\ k$ means the ratio of the correct predicted samples to the number of samples selected by ALR, therefore, the precision of $Top\ k$ is the same as the recall of $Top\ k$. We choose to only evaluate to Top 100 because the number of adversarial samples that each original malware can generate is not equal, but almost all original malware can generate more than 100 adversarial samples.

According to Figure 5.5, both methods show a growing trend as the k is bigger. However, MVML active learning consistently achieves better recall results than the random selecting method or only using the best single LTR models. In all subfigures of Figure 5.5, the highest recall result and average recall values of MVML models synthesizing are always higher than using the other two methods.

Evaluating which obfuscation and combination strategies are more likely to produce strong evasive adversarial samples. As shown in Table 5.8 and 5.9, the impacts of combination obfuscation strategies for VirusTotal's antivirus engines on obfuscated malware samples are more pronounced than using a single obfuscation strategy. The numbers in the strategies column indicate the single obfuscation strategy in Table 5.1. Among all the most evasive adversarial samples, none of them are obfuscated using a single obfuscation strategy. We can also tell that combinations with three types of strategies will generate the most evasive adversarial samples. In particular, combinations which contain *String encryption*, *Class renaming*, *API injection* and *API reflection* have significantly influences on the antivirus engines.

The class/method/variable name is often obfuscated into an arbitrary identifier composed of both uppercase and lowercase letters and numbers, which can be used to evade machine-learning-based detectors that rely on recognizing and matching identifiers. After obfuscation, these samples are detected as positive by four antivirus engines, a remarkable decrease. Class, method, and variable names are usually obfuscated with arbitrary identifiers composed of uppercase and lowercase letters and numbers. These identifiers can be used to evade machine learning-based detectors that rely on identifying and matching identifiers.

Table 5.8: Top 10 strategies in the testing set ranked by ALR

Top 10 strategies	Decrease of VirusTotal's positive antivirus engines
STRING_RESOURCE_IMAGE_RESOURCE_XML_API_INTER_API_REFLECTION	76.92%
STRING_PCM_API_INTER_BEN_PERMISSION_API_REFLECTION	73.68%
STRING_PCM_RESOURCE_IMAGE_API_INTER_BEN_PERMISSION_API_REFLECTION	73.33%
STRING_PCM_RESOURCE_XML_BEN_PERMISSION_API_REFLECTION	70.96%
STRING_PCM_RESOURCE_XML_API_INTER_BEN_PERMISSION_API_REFLECTION	69.69%
STRING_VARIABLE_PCM_RESOURCE_IMAGE_API_INTER_BEN_PERMISSION	68.75%
STRING_PCM_RESOURCE_IMAGE_API_INTER_BEN_PERMISSION_API_REFLECTION	67.56%
STRING_VARIABLE_PCM_BEN_PERMISSION_API_REFLECTION	61.11%
STRING_PCM_RESOURCE_IMAGE	55.00%
STRING_PCM_RESOURCE_IMAGE_API_REFLECTION	51.70%

Table 5.9: Bottom 10 adversarial samples in the testing set ranked by ALR

Bottom 10 strategies	Decrease of VirusTotal's positive antivirus engines
STRING_VARIABLE_BENIGN_CLASS_RESOURCE_IMAGE_BEN_PERMISSION	26.67%
RESOURCE_IMAGE	22.43%
VARIABLE_BENIGN_CLASS_RESOURCE_XML	21.25%
RESOURCE_XML	19.44%
RESOURCE_XML	18.21%
VARIABLE_RESOURCE_XML	17.50%
VARIABLE_RESOURCE_XML	17.27%
BENIGN_CLASS	12.90%
VARIABLE_RESOURCE_IMAGE	7.89%
VARIABLE_BENIGN_CLASS_BEN_PERMISSION_API_REFLECTION	2.44%

5.5 Threat To Validity

5.5.1 Threats to External Validity

The community claimed that VirusTotal's label given to the malware is not always stable [128–130], which results in a substantial impact on threshold-based labeling strategies. Because VirusTotal replace or update the scanners embedded in the website. Therefore, the threshold used to produce the most accurate label may change. Alternatively, using outdated or incorrect thresholds will change the proportion of benign programs and malware even in one dataset, effectively producing changing detection results, as shown by community's study [131, 132]. As a result, the samples with the strongest evading ability may produce different detection results after a while.

5.5.2 Threats to Internal Validity

In addition, we also found that the same single and combined obfuscation strategies for different original Android malware, the corresponding adversarial samples have different evading capabilities. Thus, no fixed obfuscation strategy or a combination obfuscation strategy will produce the strongest evading samples for all Android malware. The reason is apparent: different obfuscation strategies produce different detection avoidance effects for different types of Android malware, and the causal link between them needs more study.

Besides, the model suffers form the unstable performance of the active learning method. Active learning is to select from samples according to the selection strategy specified by itself. In this process, the strategy and data samples are two very important factors that affect performance. For very redundant data sets, active learning is often better than random sampling, but for data sets with very diverse sample data and low redundancy, active learning sometimes has a worse effect than random sampling. Also, the distribution of data samples also affects different active learning methods, such as uncertainty-based methods and diversity-based methods.

5.5.3 Threats to Construct Validity

Which evaluation metrics of prediction performance to choose can be a threat to the construct validity. To reduce this threat, we used the NDCG score and recall, which are widely used in other studies to assess the performance of ranking models.

CHAPTER 6

Conclusion

6.1 Weakly-labeled Android Malware Family Clustering

This chapter proposes ANDRE, a new approach to Android malware clustering that utilizes heterogeneous information including code similarity, the raw labels of AV vendors, and meta-data information to jointly learn an effective representation that embeds all malware in the network into a low-dimensional and compact hybrid feature space for effectively clustering weakly-labeled malware. The experimental results show that ANDRE achieves comparable accuracy to the state-of-the-art approaches for clustering ground-truth samples and that ANDRE can effectively cluster weakly-labeled malware which cannot be clustered by those approaches.

6.2 Investigating Code Deobfuscations on Detecting Obfuscated Piggybacked Apps

This chapter presents a large-scale empirical study of code deobfuscations on detecting obfuscated Android piggybacked apps. First, we generated obfuscated piggybacked apps by using ten obfuscation strategies on 1,399 Android piggybacked apps. Using the obfuscated apps generated, we then used two commonly used deobfuscation tools, i.e., Simplify and Deguard, to produce deobfuscated piggybacked apps. The total number of generated obfuscated and deobfuscated piggybacked apps was 11,378. Next, we have conducted an empirical evaluation of the generated obfuscated and deobfuscated piggybacked apps on three types of Android anti-malware detectors, including commercial anti-malware products (VirusTotal), machine learning-based detectors (Drebin and CSBD), and similarity-based detectors (Androguard and SimiDroid). The analysis was performed on two aspects: the impact of code obfuscations and deobfuscations. Along with seven findings, our results provided valuable insights into the design of Android malware detectors against obfuscations and deobfuscations.

6.3 Active Learning Based Ranking for Adversarial Android Malware

This chapter proposes a new approach to generate and precisely select the most potent and evasive adversarial malware variants. We extract the adversarial samples' representation using both unstructured and structured features. We then utilize a new MVML active learning method to produce a training set to facilitate building LTR models. The experiment results show the effectiveness of our approach in generating adversarial samples for the robustness testing antivirus tools. Also, the selected adversarial samples are proved to have the solid evading capability that can be used to evaluate antivirus engines. To conclude, the active learning method can optimize the adversarial sample generation process and effectively select the most evasive testing samples.

CHAPTER 7

Research Plan and Ethical Issues

7.1 Research Plan

Based on the key results obtained in this thesis, there are several future directions that can be explored to further develop, including the working principal of VirusTotal, application of active learning method for the machine learning-based Android malware detection.

7.1.1 VirusTotal's Labelling Policy

When training a newly developed malware detection model, the researchers relied on a threshold-based labelling strategy based on online platforms such as VirusTotal. Because the impracticality of generating accurate labels through manual analysis and the lack of reliable alternatives forced researchers to leverage VirusTotal to label applications. The dynamic nature of the platform makes these labelling strategies unsustainable for long periods of time, resulting in inaccurate labelling. Using incorrectly labeled applications to train and evaluate malware detection methods can severely undermine the reliability of their results, leading to either abandoning promising detection methods or adopting inherently insufficient detection methods.

7.1.2 Active learning strategies with good versatility

Active learning is to use fewer training samples to obtain a better-performing classifier, query the most useful unlabeled samples through a certain algorithm, and hand them over to experts for marking, and then use the queried samples to train classification model to improve the accuracy of the model. It is a data selection strategy, so a more generalized active learning strategy must be required in practical applications. However, the current active learning strategies are difficult to transfer between different domains and different tasks. Active learning strategies with good versatility could be a very promising direction for future research plan.

7.2 Ethical Issues

I have read HREC Guidelines for Undergraduate and Postgraduate Students carefully. According to this guidance, this research does not need HREC approval. The theory, model, algorithms and experiments are not involved human. The experimental datasets are from open source platforms.

Bibliography

[1] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Av-class: A tool for massive malware labeling. In *RAID*, pages 230–253. Springer, 2016.

[2] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *MSR*, pages 425–435, 2017.

[3] Xuxian Jiang. Security alert: New stealthy android spyware-plankton-found in official android market. *Department of Computer Science, North Carolina State University, URL: http://www. csc. ncsu. edu/faculty/jiang/Plankton*, 2011.

[4] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *TIFS*, 13(8):1890–1905, 2018.

[5] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, Qinghua Zheng, and Ting Liu. Frequent subgraph based familial classification of android malware. In *2016 IEEE 27th International*

Symposium on Software Reliability Engineering (ISSRE), pages 24–35. IEEE, 2016.

[6] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *SIGSAC*, pages 1105–1116. ACM, 2014.

[7] Virustotal. `https://www.virustotal.com/home/upload`. Accessed 2018.

[8] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *TPAMI*, 35(8):1798–1828, 2013.

[9] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Y Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[10] Shirui Pan, Jia Wu, Xingquan Zhu, Chengqi Zhang, and Yang Wang. Tri-party deep network representation. In *AAAI*, pages 1895–1901, 2016.

[11] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[12] Wenchao Yu, Cheng Zheng, Wei Cheng, Charu C Aggarwal, Dongjin Song, Bo Zong, Haifeng Chen, and Wei Wang. Learning deep network representations with adversarially regularized autoencoders. In *SIGKDD*, pages 2663–2671, 2018.

[13] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *ICML*, pages 1188–1196, 2014.

[14] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint:1901.00596*, 2019.

[15] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *CCS*, 2016.

[16] Richard Baumann, Mykolai Protsenko, and Tilo Müller. Anti-proguard: Towards automated deobfuscation of android apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 7–12, 2017.

[17] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[18] Caro naming convention. `http://www.caro.org/articles\/naming.html`.

[19] Jimmy Kuo and Desiree Beck. The common malware enumeration initiative. *Virus Bulletin*, pages 14–15, 2005.

[20] Caleb Fenton. Simplify. `https://github.com/CalebFenton/simplify`. 2016.

[21] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[22] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[23] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX*, volume 15, 2015.

[24] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE*, pages 576–587. ACM, 2014.

[25] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE*, pages 303–313, 2015.

[26] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *NDSS*, 2017.

[27] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *SIGKDD*, pages 1507–1515. ACM, 2017.

[28] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. A multimodal deep learning method for android malware detection using various features. *TIFS*, 14(3):773–788, 2019.

[29] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security*, pages 163–182. Springer, 2014.

[30] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *TIFS*, 2018.

[31] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 306–317. ACM, 2016.

[32] Médéric Hurier, Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *DIMVA*, pages 142–162, 2016.

[33] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. Android malware clustering through malicious payload mining. In *RAID*, pages 192–214, 2017.

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[35] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *SIGKDD*, pages 701–710, 2014.

[36] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *SIGKDD*, pages 855–864, 2016.

[37] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.

[38] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[39] R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler, and Yoshua Bengio. Learning deep representations by mutual information estimation and maximization. *arXiv preprint arXiv:1808.06670*, 2018.

[40] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *TIFS*, 9(1):99–108, 2013.

[41] Min Zheng, Patrick PC Lee, and John CS Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *DIMVA*, pages 82–101. Springer, 2012.

[42] Sudipta Ghosh, SR Tandan, and Kamlesh Lahre. Shielding android application against reverse engineering. *International Journal of Engineering Research & Technology*, 2(6):2635–2643, 2013.

[43] Rowena Harrison. Investigating the effectiveness of obfuscation against android application reverse engineering. *Royal Holloway University of London, Tech. Rep. RHUL-MA-2015-7*, 2015.

[44] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *ESORICS*, pages 293–311. Springer, 2015.

[45] Matteo Pomilia. A study on obfuscation techniques for android malware. *Sapienza University of Rome*, page 81, 2016.

[46] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[47] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE S&P*, pages 95–109. IEEE, 2012.

[48] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Ming Fan, Eryue Zhuang, and Zijiang Yang. Exploiting thread-related system calls for plagiarism detection of multithreaded programs. *Journal of Systems and Software*, 119:136–148, 2016.

[49] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Eryue Zhuang, Ming Fan, and Zijiang Yang. Reviving sequential program birthmarking for multithreaded software plagiarism detection. *IEEE Transactions on Software Engineering*, 44(5):491–511, 2017.

[50] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329, 2016.

[51] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 358–369. IEEE, 2017.

[52] Anatoli Kalysch, Oskar Milisterfer, Mykolai Protsenko, and Tilo Müller. Tackling androids native library malware with robust, efficient and accurate simi-

larity measures. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–10, 2018.

[53] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. On tracking information flows through jni in android applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 180–191. IEEE, 2014.

[54] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828, 2018.

[55] Tapasya Patki. Dasho java obfuscator, 2008.

[56] Eric Lafortune. Proguard. *http://proguard. sourceforget. net/*, 2004.

[57] Y Piao, JH Jung, and JH Yi. Deobfuscation analysis of dexguard code obfuscation tool. In *Proceedings of the International Conference on Computer Applications and Information Processing Technology*, volume 107, page 110, 2013.

[58] Allatori Android Obfuscator. Android obfuscation-java obfuscator, 2020.

[59] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *International Conference on Security and Privacy in Communication Systems*, pages 172–192. Springer, 2018.

[60] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 26–36. IEEE, 2018.

[61] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334, 2013.

[62] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–29, 2018.

[63] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE, 2016.

[64] Octavian Suciu, Scott E Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14. IEEE, 2019.

[65] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[66] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[67] Hannes Schulz, D Titze, J Schutte, T Kittel, and C Eckert. Automated deobfuscation of android bytecode. *Department of Computer Science, The University of Munchen, Germany, July*, 2014.

[68] Yuxue Piao, Jin-Hyuk Jung, and Jeong Hyun Yi. Server-based code obfuscation scheme for apk tamper detection. *Security and Communication Networks*, 2016.

[69] David A Cohn, Zoubin Ghahramani, and Michael I Jordan. Active learning with statistical models. *Journal of artificial intelligence research*, 4:129–145, 1996.

[70] David D Lewis and William A Gale. A sequential algorithm for training text classifiers. In *SIGIR'94*, pages 3–12. Springer, 1994.

[71] H Sebastian Seung, Manfred Opper, and Haim Sompolinsky. Query by committee. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 287–294, 1992.

[72] Yoav Freund, H Sebastian Seung, Eli Shamir, and Naftali Tishby. Selective sampling using the query by committee algorithm. *Machine learning*, 28(2-3):133–168, 1997.

[73] Shlomo Argamon-Engelson and Ido Dagan. Committee-based sample selection for probabilistic classifiers. *Journal of Artificial Intelligence Research*, 11:335–360, 1999.

[74] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100, 1998.

[75] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *ICSE*, 2018.

[76] Allatori obfuscator. `http://virusshare.com`.

[77] Proguard. `http://www.guardsquare.com/en/proguard`.

[78] Dasho. `http://www.preemptive.com`.

[79] Timothy I Murphy. Android-statistics & facts. `https://www.statista.com/topics/876/android`. 2018.

[80] Avtest. `https://www.av-test.org/en/statistics/\malware/`. 2018.

[81] Pluralityvoting. https://en.wikipedia.org/wiki/Plurality_voting.

[82] Chuan Shi, Binbin Hu, Wayne Xin Zhao, and S Yu Philip. Heterogeneous information network embedding for recommendation. *TKDE*, 31(2):357–370, 2019.

[83] Yao-Hung Hubert Tsai, Paul Pu Liang, Amir Zadeh, Louis-Philippe Morency, and Ruslan Salakhutdinov. Learning factorized multimodal representations. In *ICLR*, 2019.

[84] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *AISTATS*, volume 5, pages 246–252. Citeseer, 2005.

[85] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *ICSE*, pages 1157–1168, 2016.

[86] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[87] Li Li, Tegawende F Bissyande, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *TSE*, 2019.

[88] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[89] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *ISSTA*, pages 71–82, 2015.

[90] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE*, pages 175–186, 2014.

[91] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *SANER*, volume 1, pages 403–414, 2016.

[92] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *SANER*, volume 1, pages 403–414, 2016.

[93] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[94] Virusshare. https://www.virusshare.com.

[95] Android asset packaging tool. https://developer.android.com/studio/command-line/aapt2.

[96] Topological anomaly detection. https://unsupervisedlearning.wordpress\.com/2014/08/04/topological-anomaly-detection/.

[97] Mlpclassifier. https://scikit-learn.org/stable\/modules/neural_networks_supervised.html.

[98] Svm. https://scikit-learn.org/stable/modules/svm.html.

[99] Knn. https://scikit-learn.org/stable/modules/generated/\sklearn.neighbors.KNeighborsClassifier.html.

[100] Decisiontree. https://scikit-learn.org/stable/modules\/tree.html.

[101] Randomforest. https://scikit-learn.org/stable/modules\/generated/sklearn.ensemble.RandomForestClassifier.html.

[102] Sklearn. https://scikitlearn.org.

[103] triada. https://www.kaspersky.com/blog/triada-trojan/11481/.

[104] Command and control server (c&c). https://www.trendmicro.com/vinfo/us/security/definition/command-and-control-server.

[105] Cosine similarity. https://en.wikipedia.org/wiki/Cosine_similarity. Accessed 2018.

[106] Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. All your app links are belong to us: Understanding the threats of instant apps based attacks. *ACM SIGSOFT FSE*, 2020.

[107] Apktool. https://ibotpeaches.github.io/Apktool/, 2010. Last accessed March 2019.

[108] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, 2013.

[109] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *TIFS*, 2019.

[110] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *TIFS*, 2017.

[111] Wikipedia contributors. Selenium (software) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-February-2020].

[112] Kevin Allix, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Yves Le Traon, et al. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 2016.

[113] Anthony Desnos et al. Androguard-reverse engineering, malware and goodware analysis of android applications. *URL code. google. com/p/androguard*, 2013.

[114] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017.

[115] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Automatic generation of adversarial examples for interpreting malware classifiers. *arXiv preprint arXiv:2003.03100*, 2020.

[116] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020.

[117] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[118] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov):933–969, 2003.

[119] Jun Xu and Hang Li. Adarank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 391–398, 2007.

[120] Donald Metzler and W Bruce Croft. Linear feature-based models for information retrieval. *Information Retrieval*, 10(3):257–274, 2007.

[121] Qiang Wu, Christopher JC Burges, Krysta M Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Information Retrieval*, 13(3):254–270, 2010.

[122] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[123] Qingjiu Zhang and Shiliang Sun. Multiple-view multiple-learner active learning. *Pattern Recognition*, 43(9):3113–3119, 2010.

[124] Ion Muslea, Steven Minton, and Craig A Knoblock. Active learning with multiple views. *Journal of Artificial Intelligence Research*, 27:203–233, 2006.

[125] Robert E Schapire. A brief introduction to boosting. In *Ijcai*, volume 99, pages 1401–1406, 1999.

[126] Jinho Jung, Chanil Jeon, Max Wolotsky, Insu Yun, and Taesoo Kim. AVPASS: Leaking and Bypassing Antivirus Detection Model Automatically. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, July 2017.

[127] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, pages 61–70, 2010.

[128] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer integration and performance measurement for malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 122–141. Springer, 2016.

[129] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proceedings of the Internet Measurement Conference*, pages 478–485, 2019.

[130] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 112–131. Springer, 2014.

[131] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746, 2019.

[132] Aleieldin Salem and Alexander Pretschner. Poking the bear: Lessons learned from probing three android malware datasets. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, pages 19–24, 2018.