Program analysis and machine learning techniques for mobile security

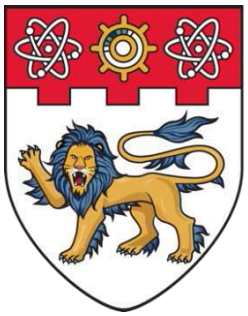
Soh, Charlie Zhan Yi

2019

Soh, C. Z. Y. (2019). Program analysis and machine learning techniques for mobile security. Doctoral thesis, Nanyang Technological University, Singapore.

https://hdl.handle.net/10356/106079

https://doi.org/10.32657/10220/47894



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

PROGRAM ANALYSIS AND MACHINE LEARNING TECHNIQUES FOR MOBILE SECURITY

SOH ZHAN YI, CHARLIE

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

2019

PROGRAM ANALYSIS AND MACHINE LEARNING TECHNIQUES FOR MOBILE SECURITY

SOH ZHAN YI, CHARLIE

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

6 MAR 2019 .....
Date


..... Soh Zhan Yi, Charlie

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

6/Mar/2019

. . . . . .

Date

. . . . . .

Chen Lihui

Authorship Attribution Statement

This thesis contains material from three papers published in the following peer-reviewed journal(s)/conference(s) where I was the first and/or corresponding author.

Chapter 3 is published C. Soh, H.B.K. Tan, Y.L. Arnatovich, and L. Wang. Detecting Clones in Android Applications through Analyzing User Interfaces. *In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension,* 163-173 (2015). DOI: 10.1109/ICPC.2015.25.

The contributions of the co-authors are as follows:

- A/Prof Tan and A/Prof Wang provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by A/Prof Tan, A/Prof Wang and Mr Arnatovich.
- I co-designed the study with Mr Arnatovich and performed all the experiments and analysis at the School of Electrical and Electronic Engineering.
- Mr Arnatovich assisted in the collection of the experimental data.

Chapter 4 is published as C. Soh, H.B.K. Tan, Y.L. Arnatovich, A. Narayanan and L. Wang. LibSift: Automated Detection of Third-Party Libraries in Android Applications. *In 2016 23rd Asia-Pacific Software Engineering Conference* 41-48, (2016). DOI: 10.1109/APSEC.2016.017.

The contributions of the co-authors are as follows:

- A/Prof Tan and A/Prof Wang provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by A/Prof Tan, A/Prof Wang and Mr Arnatovich.

- I co-designed the study with Mr Arnatovich Dr Narayanan and performed all the experiments and analysis at the School of Electrical and Electronic Engineering.
- Mr Arnatovich and Dr Narayanan assisted in the collection of the experimental data.

Chapter 6 is published as C. Soh, A. Narayanan, L. Chen, Y. Liu and L. Wang. Apk2vec: Semi-Supervised Multi-view Representation Learning for Profiling Android Applications. *In 2018 IEEE International Conference on Data Mining* 357-366, (2018). DOI: 10.1109/ICDM.2018.00051.

The contributions of the co-authors are as follows:

- A/Prof Chen and A/Prof Wang provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by A/Prof Chen and Dr Narayanan.
- I co-designed the study with Dr Narayanan and performed all the experiments and analysis at the School of Electrical and Electronic Engineering.
- Dr Narayanan and A/Prof Liu assisted in the collection of the experimental data.

6 MAR 2019 .....
Date

..... Soh Zhan Yi, Charlie

Acknowledgements

The years I spent on my PhD study in NTU have without doubt been the most rewarding part of my life. Campus life, research challenges, working with professors and colleagues have equipped me with necessary skills to thrive. I am eternally grateful to all those who provided me the possibility to complete this report.

I would like to express the deepest appreciation to my Ph.D advisor, Prof. Chen Lihui, who has provided me with invaluable advice, continuous support and encouragement. I am equally grateful to my Ph.D co-advisor Prof. Wang Lipo and former Ph.D advisor Prof. Tan Hee Beng Kuan for their guidance and support.

I am grateful to Prof. Liu Yang, who was in my Thesis Advisory Committee, for his valuable suggestions and comments on my research works.

I am thankful to the School of Electrical and Electronics Engineering, Nanyang Technological University (NTU) for providing me the research student scholarship. I would like to thank the Media Technology Laboratory executive, Mrs. Leow How for her support.

My special thanks go to my fellow research students, Annamalai, Yauhen and Xinyi for their valuable advice, suggestions and with whom I have exchanged ideas.

Last but not least, I wish to thank my parents and Ching Shing, to whom I am eternally indebted. I am grateful for all their love, care, encouragement, and understanding throughout this journey. I would not have come this far without their unconditional support.

Abstract

The rapid rise of the smartphone is mainly due to the availability of mobile applications (apps) that provide a wide range of functionalities. Statistics from International Data Corporation (IDC) has shown that Android is the most popular smartphone platform over the past few years. Due to the openness and low threshold of entering the Android app market, it has the largest collection of mobile apps. Unfortunately, it's popularity also attracts the unwanted attention of cyber-criminals. McAfee has reported that the number of threats families found in Google Play has increased by 30% in the year 2017, resulting in more than 4 thousand mobile threat families and variants in their sample database. Over the past few years, concerns have been raised with respect to the increasing number of malicious and clone apps infiltrating the Android markets. Android malware may perform a range of malicious activities e.g., leakage of sensitive information or encrypt data and lock the user out of the compromised device for ransom. On the other hand, clone apps are repackaged apps that steal revenue from the original developer of the popular apps.

Despite the advances in mobile security, the detection of malicious and clone mobile apps is non-trivial and remains an open problem. In order to differentiate these adversary apps from benign apps, an in-depth understanding of the apps is required. However, due to the arms race between the adversary apps and the detection algorithms, the adversary apps are constantly evolving and becoming more sophisticated. Hence, new and more effective algorithms are imperative. In this thesis, we address the problem of Android security by presenting new program analysis and machine learning approaches we have developed for the vetting of Android apps. The achievements made in this thesis are as follow:

1. We develop a novel approach to detect clone in Android apps by analyzing runtime user interface (UI) information. We take advantage of the unique multiple entry points characteristic of Android apps, to collect the UI information

efficiently and avoid the need for the tedious process of having to create multiple sets of relevant inputs to navigate through the entire app. An inherent advantageous characteristic of our approach is that is it resilient to code obfuscation since semantics preserving obfuscation techniques do not have any influence on runtime behaviors. The evaluation of the proposed approach was performed on a set of real-world Android apps and the results reveal that it can achieve low false positive rate and false negative rate. We further analyze the results and observe that our approach is effective in detecting different types of repackaging attacks.

2. Third-party libraries (TPLs) are commonly found in Android apps and various reports on the privacy risks and security threats that are brought about by them have surfaced. In addition, there have also been multiple complains of TPLs hindering various program analysis tasks, such as clone detection, static taint analysis, etc. Understandably, since TPLs are typically used as it is, it may include an abundance of unnecessary code to the host app and may dilute the features and increase the complexity of the code analysis, thus affecting the accuracy and scalability of the tasks. A typical and straightforward solution for identifying and excluding the TPLs is to match the name of the packages in the app to a whitelist which holds a list of known TPL package names. However, these whitelists are vulnerable against the commonly employed renaming obfuscation technique and given the fast-paced ecosystem, it is also difficult for the whitelist to be exhaustive. Hence, we propose LibSift, a tool which automates the process of identifying TPLs in Android apps. It identifies TPLs by analyzing the package dependencies of the app allowing it to be resilient to most of the typical obfuscation techniques.

3. In recent years, several promising Machine Learning (ML) based Android malware detection approaches that achieve remarkable results have been proposed in the literature. Most of these approaches are built upon the batch learning model, where a common assumption of such model is that the underlying probability distribution of the observed characteristics belonging to the data source (i.e., malware samples) is stationary. However, apart from the arms race, mobile apps are constantly evolving due to several factors such as environmental changes and adding features. These evolutions cause the distribution of the population to change over time. Moreover, in the real-world use case, the malware detection models are trained on the existing dataset and used to predict forthcoming samples that stream in. Consequently, in the face of malware evolution, the detection accuracy of the model in the real-world scenario will degrade over time. We perform

a systematic study to examine the challenges faced by state-of-the-art ML based Android malware detection approaches in the presence of concept drift. Furthermore, we propose and evaluate the modification to the approaches that may help them to overcome their limitations in handling streaming features, samples, and classes.

4. Traditionally, extensive feature engineering effort is spent to develop a solution for each of the critical Android program analytics tasks to address the multitude of problems have plagued the Android ecosystem. Hence, we aim to build holistic behavioral profiles of Android apps with rich and multi-modal information (e.g., incorporating several semantic views of an app such as API calls, system calls, etc.). Such profiles could be used to address various downstream program analytics tasks such as malware detection, clone detection, and app recommendation etc. Towards this goal, we design a data-driven Representation Learning (RL) framework named apk2vec which incorporates various state-of-the-art RL paradigm such as semi-supervised, multiview and hash embedding techniques to automatically generate succinct and versatile representation (*aka* profile or embedding) for Android apps.

In sum, this thesis proposes three methods and one empirical study with suggestions for Android apps analysis. We address four specific issues that plague Android security, namely, app clone detection, third-party library detection, malware detection, and concept drift. We do so through leveraging on techniques such as program analysis, Machine Learning and Deep Learning.

Contents

List of Figures

List of Tables

Symbols and Acronyms

Symbols

P	Set of query points
ρ	Radius
σ	Similarity threshold
N	Set of nodes in DG
E	Set of edges in DG
W	Set of weights of the edges in DG
δ	Dimension of feature vector
$\mathbb{A}$	Set of *apk*s
$\mathbb{L}$	Set of *apk*s' labels
λ	Set of node labels
$\mathcal{T}$	Vocabulary of words
$\vec{w}$	Input word embedding
$\vec{w'}$	Output word embedding
$\mathbb{D}$	Set of documents
B	Number of hash buckets
K	Set of token ids
$\mathbb{G}$	Set of graphs
k	Number of hash functions
$(\mathcal{E}$	Number of epochs
α	Learning rate
$\Phi^{\mathbb{A}}$	*apk* embeddings

Acronyms

ML	Machine Learning

AVD	Android Virtual Device
TPL	Third-Party Library
API	Application Programming Interface
RL	Representation Learning
OS	Operating System
ARI	Adjusted Rand Index
NMI	Normalized Mutual Information
PA	Passive-Aggressive
CER	Cumulative Error Rate
AMD	Android Malware Dataset [4]
IDC	International Data Corporation
UI	User Interface
DEX	Dalvik Executable
APK	Android package
VM	Virtual Machine
UID	User ID
LSH	Locality-Sensitive Hashing
ADB	Android Debug Bridge
AVD	Android Virtual Device
FPR	False Positive Rate
FNR	False Negative Rate
SI	Similarity Index
PDG	Program Dependency Graph
P_kDG	Package Dependency Graph
SVM	Support Vector Machine
RF	Random Forest
CFG	Control Flow Graph
CSBD	CFG-Signature Based Detection
IG	Information Gain
ICFG	Inter-procedural Control Flow Graph
DG	Dependency Graph
ADG	API Dependency Graph
SDG	Source/sink Dependency Graph
P_mDG	Permission Dependency Graph
WLK	Weisfeiler-Lehman Kernel

FSG	Frequent Subgraphs
HIN	Heterogenous Information Network
SGD	Stochastic Gradient Decent
OOM	Out of Memory
ROC	Receiver Operating Characteristic
AUC	Area Under the ROC Curve
CNN	Convolution Neural Network
RNN	Recurrent Neural Network

Chapter 1

Introduction

The Android Operating System (OS) dominates the worldwide smartphone market share, holding as much as up to to 85% of the market share [5]. At its peak, the official Android application (app) market from Google , namely, Google Play, has hosted up to 3.5 million apps [6]. One of the main reasons for the large amount of developers to be drawn to Android could be contributed to the ease of creating an Android app and uploading it to the numerous app markets. A plethora of free tutorials and tools for developing Android app are widely available online. There are also multiple libraries (including TPLs) available to ease the developers' job. Consequently, the Android market is filled with millions of apps providing a wide range of functionalities that become an indispensable part of the users' life. The users use these apps for their daily communications, such as email, social media, online banking and online purchases. As a result, smartphones hold a wealth of private information. For example, a typical smartphone will hold private information such as call log, SMS, email, location, browsing history, banking information etc. The popularity together with the plethora of private information have drawn the extensive attention of the malicious authors and plagiarists. Recent studies have shown that privacy leakage are commonly found in Android apps [7–9] and privacy leakage introduced by TPL are prevalent [10]. Aggravating the situation, the malicious apps created by the adversaries have been constantly evolving to increase their propagation rate and to evade detection [11–16]. In sum, the Android app market is plagued with different types of adversarial apps and there is an imperative need for the development of effective and scalable techniques to detect them.

1.1 Motivations and Goals

In particular, based on our observations of the Android ecosystem, we identify the following problems to be addressed in this thesis:

- **P1. Efficient and obfuscation resilient clone detection.** App cloning is a serious threat to the Android ecosystem in multiple ways. Firstly, a poorly repackaged app may lead to poor app usage experience for the users. In more severe cases the plagiarist may include malicious payload that may leak the users private information or even perform operations that will incur costs for the users. Secondly, app clones may reduce the revenue and reputation of the developers. Lastly, with the users and developers having negative experience, the health of the market will be negatively impacted. Therefore, it is critical to develop an accurate and efficient clone detection approach.

- **P2. Automated third-party libraries detection.** The presence of TPLs in Android apps affects the accuracy of static code-based feature app clone detection, because the same TPL can appear in many different apps and the TPLs in the original app can be easily replaced in the app clone. Similarly, the presence of TPLs dilutes the features of static code-based malware detection and affects the detection result. Besides app clone and malware detections, TPLs have also been reported to contribute greatly to the overall time and resource consumption of static code analysis task. Hence, effective third-party libraries detection is crucial for Android program analytics tasks.

- **P3. Machine learning based Android malware detection in face of concept drift.** Machine Learning based Android malware detection approaches are typically built upon the batch learning model, where a common assumption of such models is that the underlaying probability distribution of the observed characteristics belonging to the data source (i.e., malware samples) is stationary. On the contrary, mobile apps in the real-world are constantly evolving and these evolutions cause the underlaying distribution of the population to change overtime. In practice, the malware detection models are trained on existing dataset and used to predict the forthcoming samples that stream in. In face of malware evolution, the detection accuracy of the model in real-world scenario will degrade overtime.

- **P4. Holistic behavior profiles of Android apps.** Due to the multiple issues that plagued the Android app market, numerous program analytics tasks are required to be performed to address them. Typically, ML algorithms that work on vectorial representations are employed in these tasks. Therefore, having high quality behavior profiles (representation) of the apps are of paramount importance in achieving good performance in the aforementioned tasks.

These problems motivate the main work in this thesis. They are non-trivial to solve as there exist many technical challenges. The challenges for each problem are listed in the following, respectively.

- **C1.1. Obfuscation.** Android apps typically employ certain form of obfuscation and doing so decrease the performance of static analysis techniques.
- **C1.2. Scalability.** Most obfuscation techniques are semantic preserving and do not affect dynamic analysis. However, dynamic analysis suffers from scalability issues due to the need for the execution of the entire app.
- **C2.1. Dynamic ecosystem.** Owing to its popularity, the Android ecosystem is very dynamic and a whitelist of TPLs that can be used to identify the TPLs in an app needs to be updated very rapidly.
- **C2.2. Renaming obfuscation.** It is common for Android apps to employ obfuscation technique which renames the packages, classes and methods, thus, rendering the whitelist ineffective.
- **C3.1. Concept drift.** In order to stay relevant in face of concept drift, a malware detection and malware family classification tool should be able to administer any previously unseen features, samples and malware family. However, due to the sheer volume of Android apps, it is impractical to frequently retrain the detection model.
- **C4.1. Handcrafted features.** Recent studies [17–26] have shown that graph representation is ideally-suited for app profiling. A typical solution to representing graph as vectors is to use graph kernels which leverage on graph substructures, such as graphlets, shortest-paths etc. However, this often leads to building non-smooth, sparse and very high dimensional graph embeddings, thus yielding suboptimal results.

- **C4.2. Supervised vs unsupervised learning.** Fully supervised embedding methods typically require an exceedingly large volume of labeled data to learn meaningful embeddings and the resulting embeddings are not transferable to other tasks. While unsupervised embedding methods do not exhibit such limitations, they are unable to leverage on any available labeled data.

- **C4.3. Scalability.** Alternatives to graph kernels are data driven graph embedding approaches that provide better generalization capabilities. However, due to the exceedingly large number of parameters, graph embedding methods typically have poor memory and time scalability.

- **C4.4. Integrating multiple views.** Features providing different semantic views can be extracted from Android apps. Effectively integrating such multivariant information while maintaining scalability is challenging, yet critical in building holistic and versatile embeddings capable of catering to a variety of tasks.

1.2 Main Work and Contributions

Four main works are conducted to solve the aforementioned problems and technical challenges as shown in Figure 1.1. First, we propose to detect Android app clones through analyzing their user interfaces. Secondly, as mentioned in several studies [30, 39, 51, 67, 72, 75] the use of TPLs have negative impact on Android program analysis tasks such as clone detection. Therefore, we propose `LibSift`, an automated tool that performs TPLs detection based on package dependencies. Third, we evaluate the performance of state-of-the-art Android malware detection approaches in the face of concept drift. We also propose solutions to address the limitations of the approaches. Finally, to avoid the extensive feature engineering to address each Android program analytics tasks respectively, we propose apk2vec that builds holistic and task agnostic app behavior profiles that are capable of catering to various downstream tasks. We elaborate these works as follows:

1. **Clone detection through analyzing user interfaces.** In recent years, the repacking of app has become a major concern in the Android ecosystem. Traditionally, work on software clone detection mainly focuses on code-based

analysis. However, such methods are vulnerable to advanced obfuscation techniques which are very common among Android apps. They also face potential scalability issues, due to the large amount of apps and billions of opcodes that need to be analyzed when performing app clone detection across the numerous app markets. To this end, we propose a novel technique of detecting Android application clones based on the analysis of user interface (UI) information collected at runtime. Doing so provides two main advantages. Firstly, by leveraging on the multiple entry points feature of Android applications, the UI information can be collected easily without the need to generate relevant inputs and execute the entire application. Secondly, our technique is obfuscation resilient since semantics preserving obfuscation techniques do not affect runtime behaviors.

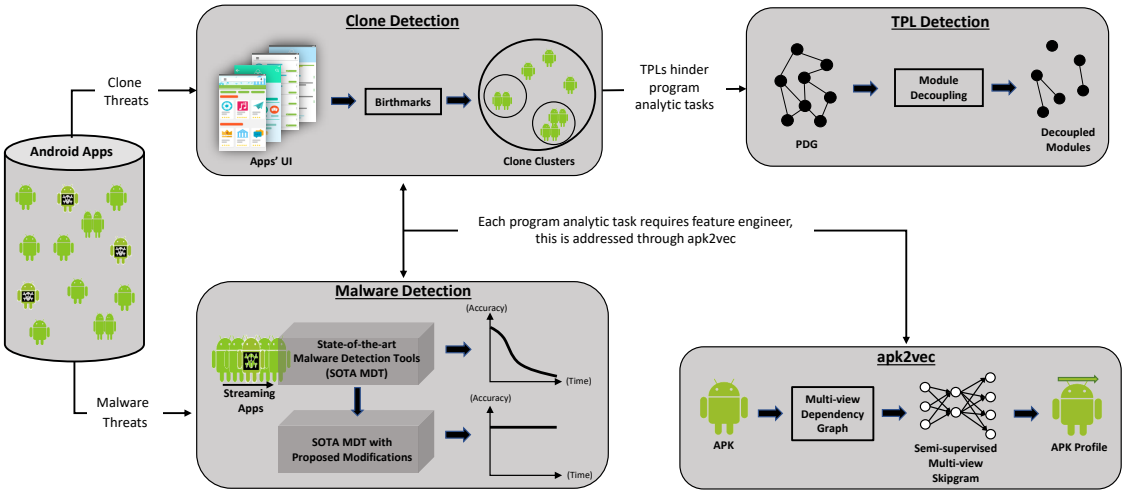


FIGURE 1.1: Overview of the thesis

Contributions. We make the following contributions in this work:

- We propose an novel approach to detect clones in Android app based on dynamic features obtained from runtime UI information. To the best of our knowledge, we are the first to leverage on such feature to address Android app repacking problem.
- We leverage on the unique multiple entry points characteristic of the Android apps to avoid the non-trivial generation of relevant inputs to navigate the entire app.
- We evaluate our approach on a set of real-world top popular Android apps from different categories and across four different Android markets.

The result reveals that our approach has low false positive and false negative rates.

- We also evaluate our approach on a collection of clone sets detected by code analysis based approach. The results show that our approach can effectively identify various types of repacking attacks.

2. **Third-party libraries detection based on dependency graph.** Typically each Android app encompasses several TPLs and there has been increasing concerns on the potential security threats and privacy risks that TPLs may induce to the host app. Furthermore, several recent studies has considered excluding the TPLs from their program analysis tasks as the presence of TPLs may dilute the prominent features and affect their model's accuracy. A majority of these studies employ a whitelist to identify the TPLs to be excluded from their analysis. However, due to the dynamic ecosystem the whitelists are often severely incomplete. They are also vulnerable to the common obfuscation technique which renames the packages in the apps. In this work, we propose `LibSift`, a tool which detect TPLs in Android apps automatically. `LibSift` detects TPLs by analyzing the package dependencies between the packages within the app, thus, resilient to most common obfuscations.

   Contributions. We make the following contributions in this work:

   - We propose a tool to perform automated detection of TPLs in Android apps based on package dependency graph, instead of assuming that TPLs will occur frequently in a large number of apps.
   - We design and implement a prototype of `LibSift`, and perform extensive experiments to evaluate the effectiveness and efficiency of `LibSift` in detecting TPLs.
   - We compare our approach with two state-of-the-art approaches TPL detection approaches, namely, LibRadar [2] and whitelist from Li et al. [3]. We demonstrate on a set of real-world top popular Android apps that `LibSift` can detect even not popular TPLs that are not picked out but the state-of-the-art approaches.

3. **Android malware detection in face of concept drift.** Most of the Android malware detection approaches are build upon ML technique that

is based on the batch learning model, where an important assumption is that the population distribution is stationary. However, in the real-world use case, the detection models are first trained on the existing dataset and used to predict on the forthcoming samples that are constantly evolving. Hence, we perform a systematic study to examine the challenges faced by state-of-the-art ML based Android malware detection approaches in presence of concept drift. Furthermore, in areas where the approaches fail to perform well, we suggest and empirically evaluate the modifications to the approaches that may help them to overcome their limitations. To this end, we reimplemented two state-of-the-art approaches and use them to perform a series of empirical experiments on a large dataset of over 80K apps that spans from year 2009 to 2016. Our experiment results show that, the impact of malware evolution in such approaches is significant and requires prompt attention. In addition, our proposed modifications prove to significantly improve the performance of the approaches.

Contributions. We make the following contributions in this work:

- We reimplemented two state-of-the-art malware detection approaches with orthogonal choice in feature sets and use them to demonstrate the limitations of existing batch learning ML based malware detection and malware familial classification approaches in the face of concept drift.
- We suggest technique agnostic modifications to existing batch learning ML based malware detection and malware familial classification approaches and demonstrate using two such state-of-the-art that the suggested modifications significantly improve their effectiveness and efficiency.

4. **Building holistic and task agnostic behavior profile of Android apps.** Most Android program analytics approaches including our earlier work mentioned above, require extensive feature engineering, targeted at excelling in the respective task. For a different task, another round of feature engineering needs to be performed. As aforementioned, Android is plagued with multiple problems that require immediate attention. Analyst perform different program analytic tasks to address each problem. Building compact and versatile behavior profiles of Android apps by incorporating semantic views of different modality from the app, such as API sequences, system calls,

etc., would enhance the capability of the profile to be applicable in various downstream program analytics tasks such as clone and malware detection, malware familial clustering and app recommendation. Thus, eliminating the feature engineering effort necessary for each task. Towards this goal,we design apk2vec, a Representation Learning (RL) framework which incorporates various state-of-the-artlearning paradigm to automatically generate a compact representation (aka profile or embedding) for a given app.

Contributions. We make the following contributions in this work:

- We propose `apk2vec`, a static analysis based graph embeddings framework, to build task-agnostic profiles for Android apps. To the best of our knowledge, this is the first app profiling framework that has three aforementioned unique characteristics.
- We propose a novel variant of the skipgram model which train via view-specific negative sampling to facilitate integrating multi-variant learning in a non-linear manner to obtain multi-view embeddings.
- We evaluate `apk2vec` with a large dataset of real-world apps on various downstream program analytics tasks and demonstrate that `apk2vec` can outperform several state-of-the-art graph RL approaches.

1.3 Thesis Outline

The roadmap for this dissertation is as depicted in Figure 1.2. It presents the sequence of our research work and thus reveals our train of thoughts. The conference and journal publications made as part of this work are also labeled accordingly. The remaining of the thesis is organized as follows:

In Chapter 2, we present the background and related work on Android security. In particular, we introduce the preliminaries of the Android architecture, followed by the status quo of Android clone detection and malware detection. The relevant ML algorithm and paradigm are also discussed here.

Chapter 3 presents our work on Android app clone detection. We propose to explicitly start the activities within the app and extract their runtime UI information. We then convert the UI structure information into feature vectors. To detect the

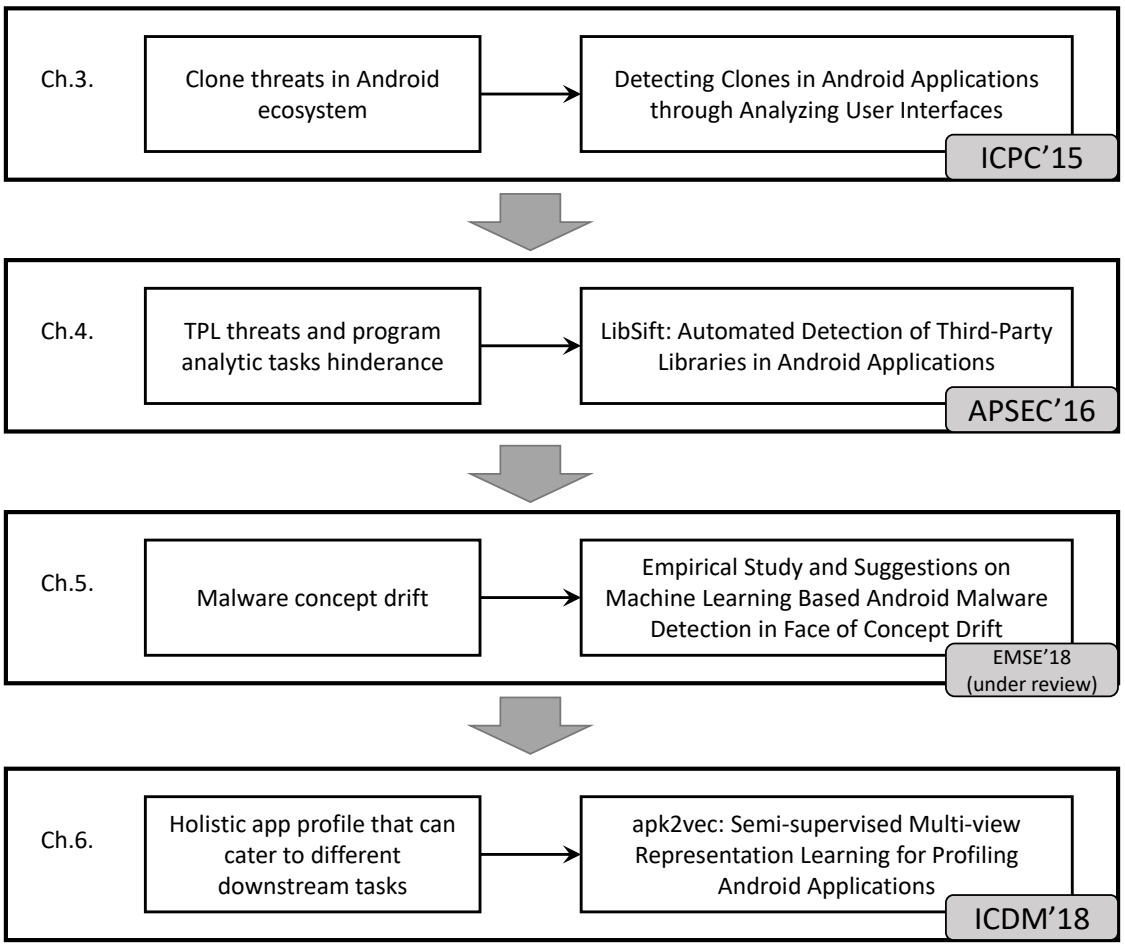


FIGURE 1.2: Roadmap of the thesis

app clones, we apply LSH algorithm on the vectors to find the near neighbours before applying Hungarian algorithm to find the best match resulting in the highest overall similarity. We also present the prototype implementation of our approach and demonstrate its accuracy and scalability. We further examine the ability of our approach to detect different types of app clone attacks.

In Chapter 4, we present `LibSift`, a tool to automatically detect TPLs in Android apps. `LibSift` detects TPLs based on package dependencies that are resilient to most common obfuscations. We demonstrate that `LibSift` can detect even the less popular libraries that are not detected by two of the state-of-the-art approaches.

In Chapter 5, we present a systematic and empirical study to examine the challenges faced by state-of-the-art ML based Android malware detection approaches in presence of concept drift. In addition, we also suggest and empirically evaluate the

modifications to the approaches that may help them to overcome their limitations in handling streaming features, samples and classes.

In Chapter 6, we present apk2vec, a semi-supervised Representation Learning (RL) framework named apk2vec to automatically generate a compact representation (aka profile or embedding) for a given app. Evaluations with more than 42,000 apps demonstrate that apk2vecs app profiles could outperform state-of-the-art solutions in four app analytics tasks namely, malware detection, familial classification, app clone detection and app recommendation.

In Chapter 7, we conclude the work in the thesis and propose the potential directions in the future.

Chapter 2

Background and Preliminaries

This chapter provides background on the Android ecosystem and preliminaries on the defense techniques. In particular, Section 2.1 presents the architecture of the Android OS. Section 2.2 focuses on the Android security mechanism. Section 2.3 reviews the Android app fundamentals, such as apps development, app components and inter-component communication. Section 2.4 discusses Android markets, the main distribution channel of Android apps. Section 2.5 and Section 2.6 provide an overview of app clone and TPLs and the threats that they have introduced to the Android ecosystem, respectively. Section 2.7 discusses the defence technique against adversary Android apps.

2.1 Android Architecture Overview

The Android OS which is built upon the Linux kernel, is an open source software stack that is designed for mobile touch-enabled devices, such as smartphones or tablets, which typically have low computational power and limited battery life. As shown in Figure 2.1, the Android software stack can be generally divided into four main layers and five sections.

Linux kernel. At the base of the layers is the Linux kernel, which provides a level of abstraction between the device hardware and the layers above. It contains the drivers for hardware such as camera, display, WiFi, etc. The Linux kernel also

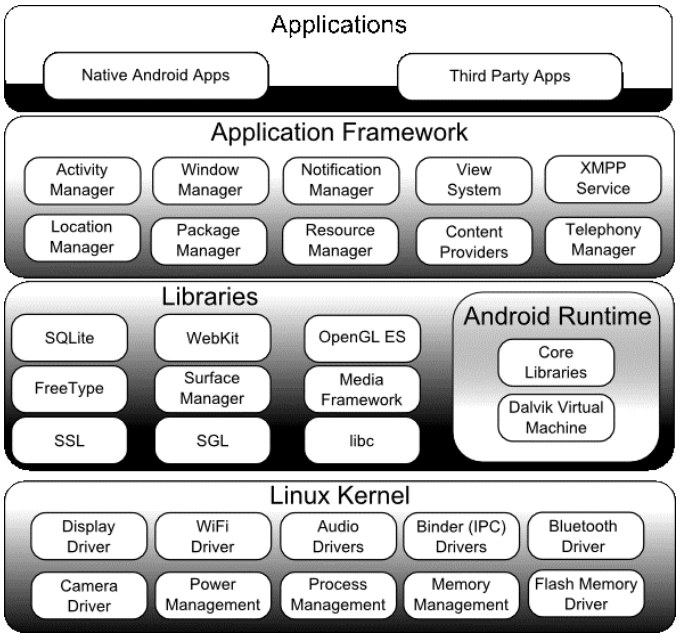


FIGURE 2.1: Android software stack. [1]

provides interfaces from the underlying functionalities such as low-level memory management and multitasking.

Libraries. Above the Linux kernel layer, at the second layer is a set of native libraries. These libraries are responsible for media playback, browser support and SQLite database support, etc.

Android runtime. Along with the libraries section, the second layer also contains the Android runtime section, which consists of the core libraries and Dalvik VM or Android Runtime. The set of core libraries allows the developers to develop Android apps in the Java programming language. The Dalvik VM is a register based VM that is optimized for mobile devices as it can provide good performance with less memory consumption. To improve the runtime performance, from Android version 5.0, Google introduced Android Runtime as a replacement for Dalvik VM. The Android Runtime practices Ahead-of-Time (AOT) compilation as contrast to Dalvik VM Just-in-Time (JIT) compilation. The replacement of Dalvik VM with Android Runtime is generally a trade-off between runtime performance and more storage required for installation.

Application framework. The third layer is the application framework which contains a set of services, in the form of Java classes, that Android app developers can make use of in their apps.

Applications. At the top of the stack is the Application layer, where the system apps and third-party apps reside. These apps use the services provided by the application framework and run within the Android runtime.

2.2 Android Security Framework

The Android security mechanism which helps to keep the platform and ecosystem safe is built upon the Linux kernel. Since the release of Android, Google has consistently issue updates to improve Androids security. In this section, we briefly highlight the key Android security mechanisms.

Application isolation. The Android security philosophy requires that each Android app runs in a sandbox, isolated from the other apps. That is, in the Android device, each app runs in its own process, in its own instance of Dalvik Virtual Machine (VM). Similar to the Linux kernel that Android is built on, when a third-party app is being installed on the Android device, it is assigned a user ID (UID). The UIDs are used to isolate the apps. App developers have the option to allow their apps that are signed with the same key to share the same UID, and thus, run in the same sandbox. Furthermore, based on the UID, each app is also assigned a private directory in the file system. Apps sharing the same UID, may access each other's files. It is worth noting that, the TPLs that are included in the app also run in the same process as the host app and have access to the files of the host app.

Privilege separation. Android follows the Principle of Least Privilege [27], which means that each app should only be allowed to access the resources that it requires to complete its task. Hence, in order to achieve privilege separation between the apps that are installed on the device, Android introduces a permission system. Android app developers must declare in the manifest, the permissions that are required for the app to function. Before Android version 6.0 (API level 23), users are only given the choice to either grant all the permissions requested by the app to continue the installation process or to abort the installation of the app entirely. However, from Android version 6.0 or higher, no permission will be granted at installation time, instead the permission request will be made during runtime when it is needed. Currently, official Android framework does not support the separation

privileges between the TPLs and host app, as such, TPLs are granted the same privileges as the host app.

2.3 Android Application Fundamentals

An Android app is typically a compressed file that is signed with the developer's private key before publishing on the Android market. In this section, we review the fundamental of Android apps, focusing on the app development, app components and inter-component communication. Having this knowledge is helpful in understanding the motivation and technical aspect of our work.

2.3.1 Android Application Development

Java programming language is primarily used to develop Android apps. However, instead of stopping at being compiled into Java bytecode, Android apps are compiled into the Dalvik Executable (DEX) format. This allows the Android apps to run in the Dalvik VM, which is optimised for mobile devices. The DEX file, along with all the resources necessary for the app to function, are compressed into an Android package (*apk*) archive with a .apk suffix. However, before the app can be installed on the Android device, the app is required to be digitally signed with a certificate. Currently, Android does not perform certificate authority verification and the app can be self-signed. Hence, anyone can develop and publish their Android app. After the app has been signed and published on the Android market, the *apk* file can be downloaded by the user and installed on their Android device.

AndroidManifest.xml. One of the most important file in an Android app is its AndroidManifest.xml. The Android system must be informed of the component existence in an app before it can be started. In order to do so, the app developer have to declare the components in the AndroidManifest.xml. Otherwise, the system is unable to identify the component and cannot execute them [28]. The AndroidManifest.xml also contains several other important information, such as package name and permission etc.

2.3.2 Android Application Components

Android application is made up of several components. Unlike traditional software program, Android apps possess multiple entry points. Each component can be an entry point to the app. The four basic types of components that can be found in Android apps are as follows:

Activity. An activity represents a single screen that displays the UIs for the user to interact with. An Android app typically contains more than one independent activity but work together to provide user with a smooth experience. Each activity is made up of several widgets arranged in a hierarchical structure. For our app clone detection work in Chapter 3 we focus our attention on analysing the activity component.

Service. In contrast to the activity component, the service component are typically used for long running computational intensive task, that is performed in the background, without the display of any graphic element. A service will continue to run in the background even when the app is not in the foreground.

Content provider. The content provider component provides an interface for apps to access a structured set of data. It allows content to be centralized and multiple different apps can have access to it if permitted.

Broadcast receiver. A broadcast receiver is used to register for broadcast messages that may be raised by the system or other apps. Once the event occurs, all the receivers that registered for the broadcast message will be notified.

2.3.3 Inter-Component Communication

The primary mechanism that Android app components use to communicate is known as intents. They are asynchronous messages that are used by app components to make request to other components. There are generally 2 types of intents, explicit and implicit intents.

Explicit intent. An explicit intent specifies the explicit name of the targeted component class. They are generally used to invoke components that are within the app. A typical example will be to start a new activity.

Implicit intent. On the other hand, an implicit intent does not specify the targeted component class name, instead it indicate a general action to be performed. When an implicit intent is being created, the Android system will look for the appropriate component based on its intent filter. The appropriate component may be from other apps that are installed in the same Android device.

2.4 Android Market

The Android markets are the main distribution channels for Android apps. After signing their apps with private key, the developers have the options to publish their apps on the official Android app market, third-party Android app market or even multiple app markets. The developers are required to pay a small fee to publish their apps on the official Android app market, whereas most third-party Android app markets just require the developer to create a free account.

The relaxed Android market and app signing policy together with the range of tools and tutorials freely available online, result in the increasing number of app clones we see today. Furthermore, most Android markets do not vet the apps that were published through them, but instead depend on user feedback to distinguish between good apps and bad apps. The official Android app market does include a service known as Bouncer [29], which scans the app market for malicious apps. However, it is not clear whether Bouncer has the ability to identify app clones which may or may not be malicious.

2.5 Android Application Clones

In recent years, concerns have been raised among the Android community with regards to the increasing number of app clones in the Android app markets. Zhou et al. [30] reported that among the apps in six third-party Android app markets, the rate of app clones range from 5% to 13%. This is a serious problem as the threats brought by app clones affect multiple stakeholders in the Android ecosystem. In this section, we present an overview on app clones, different types of app clones and threats that were brought by them.

Android apps are distributed in the form of *apk* files and the source code of the apps are generally not available. Despite this, the plagiarists can make use of the several reverse engineering tools [31–33], that are freely available online to disassemble the apps without much effort. Typically, the plagiarist may choose to convert the app code to an intermediate representation (i.e., smali, jimple, jasmine), Java bytecode or even Java source code. They might then modify the content of the app, which include but not limited to injecting malicious payloads and redirecting advertising revenues. Lastly, the plagiarist will sign the app clone using their own private key and publish it on the Android app market.

2.5.1 Types of Application Clones

The similarity between the code or bytecode fragment of two applications can be based on the similarity in their code statement or functionality. There are basically four types of code clones:

Type 1. Identical code fragments except for variations in layout.

Type 2. Syntactically or structurally identical code fragments except for variations in identifiers, literals, types, layout and comments, in addition to the variations in Type 1.

Type 3. Copied fragments with further modifications. Statements can be changed, added or removed in addition to Type 2.

Type 4. Code fragments that perform the same computation or functionality but implemented through different syntactic variants.

Furthermore, based on the modifications of the clones, the authors of an Android app clone detection study [34] suggested that we can generally classify the app clone attacks into three categories:

Lazy attack. Lazy attacks consist of modifications that are simpler and does not change the functionality of the app. An example of a lazy attack modification is to replace the advertisement library. Simple automated code obfuscation may also be applied to avoid detection.

Amateur attack. Amateur attacks consist of more advanced modifications that require more knowledge and effort from the plagiarist. For example, we have observed that the plagiarist may modify the TPLs used in the original app. Instances of such modifications are such as the addition of social media sharing function to the original function of the app and the change of the advertising library to redirect the revenue to the plagiarists own account. Automated or more advanced obfuscation techniques may also be applied to avoid detection.

Malware. The last form of attack is whereby the plagiarist injects malicious payload into the original functionality of the app. The malicious payload may secretly leak the users private data or perform operations that will incur cost for the user. In this case, the effort and knowledge required is higher or as much as performing an amateur attack.

2.6 Third-party Libraries in Android Applications

Code reuse is a common practice in software development. It allows software developers to leverage on pre-written code and use the available functions without writing the code. This practice simplifies the programming task and helps the developers to be more productive as they can spend more time and effort to focus on solving the unique part of their software. In today's competitive market, Android app developers typically use several TPLs to provide multiple useful features in their apps without having to spend long hours implementing it themselves. However, despite all the advantages of using TPLs, there are also several threats that come with it. Furthermore, these threats cannot be fully addressed without knowing the cause of the threat and this cannot be achieved without an effective TPLs detection tool. In this section, we briefly discuss some of these threats that motivates our study on the detection of TPLs in Android apps.

Security and privacy threats. Firstly, the TPLs enjoy the same privileges that was granted to the host app. It allows the TPLs to successfully request for additional permissions granted to the host app but are not necessary for the TPLs to complete their intended tasks. This violates the Principle of Least Privilege [27]. Secondly, TPLs are assigned the same internal storage space as the host, allowing the TPLs to access any sensitive information stored by the host app. This may

lead to the leakage of private information. Lastly, TPLs may introduce security vulnerabilities and bugs in the host app. As with any other software program, the TPLs may contain bugs and vulnerabilities and these program flaws are passed to the host app that uses such TPLs. Several existing literatures has reported that TPLs are responsible for such problems [35–37].

Program analysis. TPLs in Android apps are also known to hinder several program analysis tasks that are performed on the Android apps. Firstly, TPLs in Android apps have significant impact on the accuracy of Android app clone detection [38]. This is because, TPLs in Android apps can be easily manipulated and plagiarists are motivated to do so to evade detection or for financial gain (i.e., replace advertisement libraries). Secondly, in some analysis tasks a significant amount of time is spent analysing irrelevant code due to the use of TPLs as it typically contribute to large portion of the app code. A recent study has reported that TPLs generally make up 60% of the app code [39] and there are reports that the analysis of some app cannot be completed within a stipulated timing [40] or ran out of memory [41]. Lastly, the presence of TPLs may dilute the features that are used in malware detection. Consequently, it blurred the contrast between the benign and malicious samples and thus reducing the accuracy of the detection.

2.7 Analysis Paradigm for Android Defence

Numerous defense techniques have been proposed for Android security. These techniques are based on different analysis paradigms, that can generally be classified into two types, namely, static and dynamic analysis.

Static Analysis. The static analysis techniques are performed without executing the program. This form of analysis typically achieves better code coverage and scalability over dynamic analysis. When the source code is not available, the analysis is performed on the bytecode obtained through reverse engineering. However, the accuracy of such analysis is limited when obfuscation, dynamic code loading or reflection exist. Static analysis when performed on Android apps typically involves the extraction of features such as API sequences, permissions, instructions, control flows and data flows. The majority of the Android defense techniques, such as

clone detection [19, 39, 42] and malware detection [20, 21, 43–45], are based on static analysis.

Dynamic Analysis. Dynamic analysis on the other hand, executes the program under analysis to obtain the necessary data for performing the analysis. It typically collects the execution traces or tracks the flow of sensitive information for analysis. Dynamic analysis counteracts the limitations of static analysis, but has poor scalability and code coverage. Dynamic analysis based Android defense technique are for example, TaintDroid [8] and CrowdDroid [46].

Chapter 3

Detecting Clones in Android Applications through Analyzing User Interfaces

3.1 Introduction

The Android platform's popularity together with the ease of repackaging Android apps draws the interest of plagiarists and malware writers. Eventhough the source code of published Android apps are usually unavailable, it is straightforward to reverse engineer Android apps due to the availability of reverse engineering tools such as apktool[47], dex2jar[31] and Dare[48]. With the help of these tools, the plagiarists can easily repackage any *apk* file downloaded from any Android market by first disassembling it then modify its content before repackaging and signing it as their own app. This repackaging attack is also commonly known as cloning.

The presences of repackaged apps bring about several disadvantages and they are growing in numbers. A repackaged app which redirects advertisement revenues or contains malicious code fragments could potentially cause the legitimate developers to lose their revenues and reputations. A recent study by Gibler et al. [49] reported that assuming the users who downloaded the clones would have downloaded the original app instead, the legitimate developer could lose about 10% of the user base to the clones. Zhou et al. [50] found that 86% of the malware samples are clones of the legitimate apps. In addition to the original app market, Google Play, there

are a plethora third-party Android markets available for the developers, including the plagiarists, to upload their apps. Most of these third-party Android markets do not ensure for the quality of the apps that they host. A study by Zhou et al. [30] found that out of six third-party markets, 5-13% of the apps hosted are clones. Given the current Android market growth rate, we envision that these figures are likely to increase further.

Based on the above-mentioned it is clear that app cloning is a severe problem that needs to be addressed promptly. It not only affects the users and the developers, it also destroy the health of the Android app market.

Existing literature on repackaged app detections are typically based on static code analysis [19, 30, 51]. However, such approaches may posses several limitations. For instance, the common practice of code reuse and the usage of the advanced obfuscation techniques may have adversarial effects on such approaches [52]. Zhou et al. [50] reported that malware writers tend to employ obfuscation to avoid detection. Furthermore, since source code is generally not available, the analysis of app is usually conducted on opcodes. A study [53] on 30,000 Android apps shows that the apps with median size of 754KB have 20,555 median opcodes. With the need for analysis to be conducted across multiple markets, the number of opcodes that need to be analyzed would increase greatly, resulting in billions of opcodes to be analyzed [19].

In this chapter, we propose a novel approach for cross-market Android app clone detection based on robust dynamic software birthmarks. These birthmarks are basically a set of features which uniquely identify each of the apps. They are generated by leveraging on the view hierarchy information of the user interface (UI). The view hierarchy can be extracted in XML format when the activity is in the foreground of the system (displayed on the screen) where user can interact with it. The intuition behind our approach is based on the following observations:

- Obfuscation techniques that preserve semantics do not affect runtime behaviors. Therefore, dynamic software birthmarks have a much better obfuscation resistance compared to static software birthmarks.
- The plagiarists would want the UI design (i.e., look and feel) of the clone to resemble that of the original app so as to leverage on the popularity of the

original app. Furthermore, it may be easy to modify the position and size of the UI, but modifying the functionality of the UI requires more effort and understanding of the app code. Thus, app clones would likely have UIs which resembles the original apps' UIs.

- Differing from traditional software, Android apps have a unique feature of having multiple entry points. We can leverage on this feature to access and extract information from different components of the app directly.

Our approach provides an alternative to the traditional code-based static analysis detection approaches and does not inherit all disadvantages of dynamic analysis approaches. The main advantages of the proposed approach are as follows:

- **Obfuscation resilient.** Our approach is resilient to code obfuscation techniques, since we use dynamic UI information as features. It is obtained during runtime, thus, unaffected by semantics preserving code obfuscation techniques.
- **No input generation required.** Despite the approach being based on dynamic analysis, we leverage on the multiple entry points characteristic of Android apps to avoid having to generate inputs to navigate through the complex UI, which is a process that is hard to automate. More specifically, we extract a list of activities from the *apk*, and start each of them explicitly to obtain their runtime UI information.

Contributions. We make the following contributions in this chapter:

- We propose an approach to detect Android application clones based on the birthmarks generated from runtime UI information. To the best of our knowledge, we are the first to use runtime UI birthmarks for Android app clone detection.
- By leveraging on the multiple entry points characteristic of the Android system, our approach does not require the generation of relevant inputs for execution of the entire app.

- We evaluate our approach on a set of real-world data collected from four different Android markets. Experimental results show that our approach has low false positive and false negative rates.
- We also evaluate the effectiveness of our approach on a collection of clone set data from another research group. The results show that our approach can effectively detect different types of clone attacks.

3.2 Android User Interface

The activity component of an app contains UI components that are structured as a hierarchy of views (e.g., `Button`, `CheckBox`, `TextView`) and view groups (e.g., `FrameLayout`, `LinearLayout`, `RelativeLayout`). Each of these UI components is tasked to manage a particular rectangular space within the activity's window. A view group provides a layout of the interface for its child UI components. Therefore, it can be a parent of multiple views and view groups [54].

3.2.1 UI Information Extraction

To extract UI information for analysis, we leverage on a tool in the Android SDK known as "uiautomator". It is a testing framework that allows the developers to test the UI of their apps efficiently [55]. The tool provides a function which allows the view hierarchy of the current activity to be extracted as an XML file. Figure 3.1 shows an example of an activity's screenshot taken from an app and part of the corresponding XML file obtained via the uiautomator tool.

The extracted XML file contains runtime information on all view objects in the activity. Each view and view group is represented as a node in the XML. In addition, each node has the exact same seventeen distinct attributes with different possible values. However, if the uiautomator does not have access to a particular node, then the node will have an additional NAF attribute.

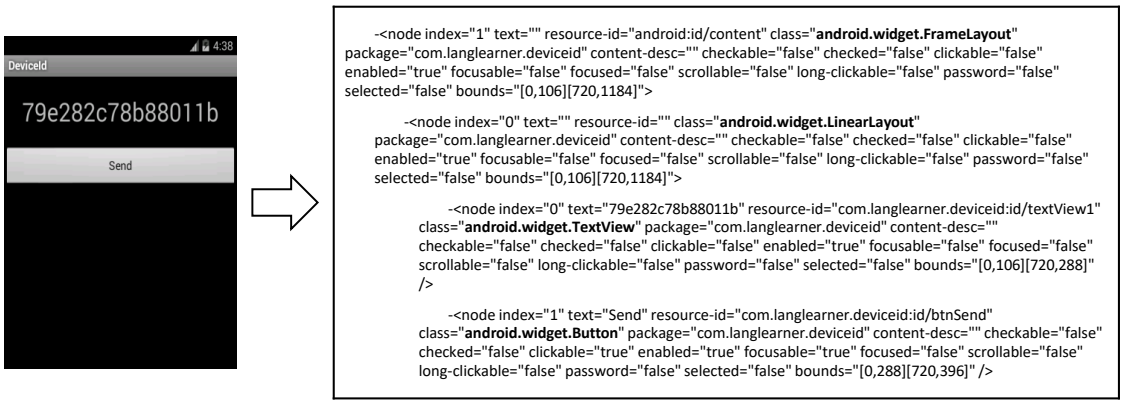


FIGURE 3.1: Screenshot of an activity with partial corresponding XML

3.3 Methodology

An overview of our proposed approach is presented in Figure 3.2. Firstly, for a given set of *apk*s, we install them on an Android emulator and extract the names of the activities. We then execute them and use uiautomator to gather their UI information in XML format. Secondly, the gathered information are passed through two filters to exclude unnecessary information from the XML files and the remaining information are used to generate birthmarks. Thirdly, we identify the near neighbors of the activities based on locality-sensitive hashing (LSH). If the near neighbor contains more than one activity from another app, we apply the Hungarian algorithm to find the pairs of activities that results in highest similarity. Lastly, we cluster the apps into groups of clone sets. We describe the details of each process in the following subsections.

3.3.1 Data Extraction

An Android app consists of multiple components. Each of these components can act as an entry point in which the system can invoke to access different functions of the app. Leveraging on said feature we use explicit intent to execute the activities in the apps. In this case, we can avoid having to generate complex sequences of inputs for the execution of each path in the app's control flow graph (CFG). However, a limitation of such method is that, in some cases, the components might not be actual entry points or may not be independent. These components cannot be accessed directly by using explicit intents.

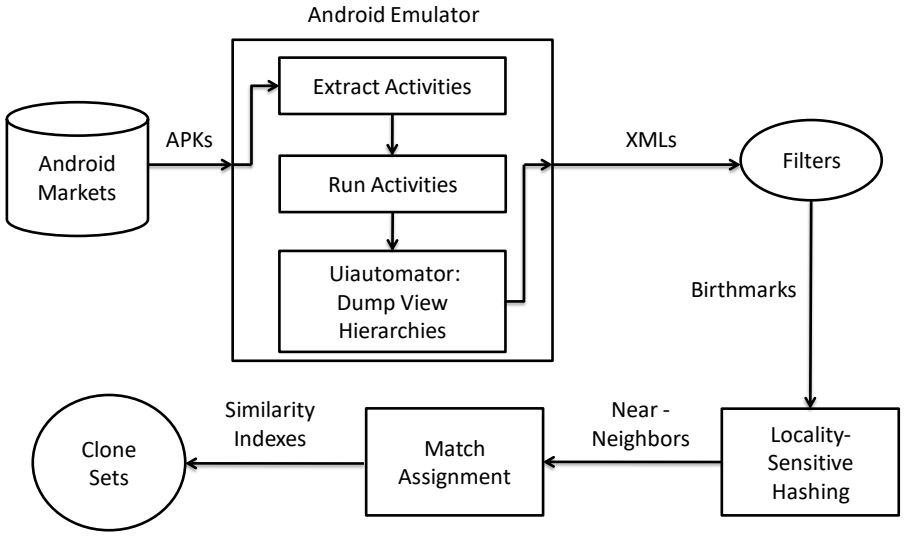


FIGURE 3.2: Overview of our clone detection approach

Nevertheless, the inaccessible components in the original app also highly likely to be inaccessible in the clone. Thus, it may not affect the similarity between this pair of apps, or at least not significantly.

Despite the risk of information lost when activities are unable to be executed using explicit intents, the advantages of eliminating the effort of generating inputs for the execution of the entire app greatly outweighs the disadvantages. The data extraction process is automated with a script in the Java programming language. In addition, we employ Android Debug Bridge (adb) and API libraries from the Android SDK [56] to interact with the *apk* files and Android virtual device. The details of the extraction process can be broken down into two main steps:

1. **Extract activity names.** It is necessary for the activities in the app to be declared in the AndroidManifest.xml. Otherwise, the Android system will not see the activities and is unable to execute them [28]. For each app, we extract the fully qualified class names of the activities and use them to start the activities explicitly. The activities in an app can be divided into two main groups, the apps activities and third-party libraries activities. Apps activities refer to the set of activities that are implemented by the app's developer and are unique to the app. On the other hand, TPLs activities refers to the set of activities that are included by TPLs. In this study, we do not consider TPLs activities, as they can be easily replaced or removed. For example, the advertisement libraries that are commonly included in various apps typically

have advertising activities and replacing the advertisement library in the app will also cause a change in the activities.

2. **Obtain XML file.** In order to extract the UI information, the downloaded *apk*s first have to be installed on an Android environment to allow the UI to be rendered. For this purpose, we use an Android emulator created from the Android virtual device (AVD) Manager. We use adb commands to install the app and use explicit intents to start the activities with their fully qualified class names. When the activity is started, we use uiautomator to dump the view hierarchy into an XML file. Thereafter, we stop the current activity and repeat the process until all the activities have been exhausted.

3.3.2 Birthmark Generation

Software birthmark is a unique characteristic that the software possesses and serves as its identity. Software birthmark can be further classified as static birthmark or dynamic birthmark. In this work, we extract the dynamic software birthmark of the app and use it for clone detection.

More specifically, we parse the XMLs obtained from the previous process to obtain birthmark for each activity in the apps. As aforementioned, each node in the XML has the exact same 17 distinct attributes with different possible values. One of the most important attribute is the "class" attribute, since its value denotes the type of view (e.g., `Button`, `TextView`, etc.) that represents the particular node. Each birthmark is a vector where each element in the vector represents the frequency count of a unique combination of the view class, selected attribute and value of the selected attribute. If all 17 attributes were used to generate the birthmarks, each vector would contain a large amount of elements. However, we observed that the attributes are not equally informative for clone detection. Therefore, improve accuracy and reduce computations, we apply two filters to exclude the unnecessary information.

Filter 1 (F1). Some attributes of the view are not informative for clone detections. Instead, they introduce noise into the birthmarks and decrease our detection accuracy. There are also attributes with string type values that can be easily manipulated and costly to compare. We analyze these attributes as follows:

- The "*index*" attribute simply represents the position of the node in the view hierarchy. The positions can be easily switched within or even out of the view group. By doing so, it would result in a different index value being assigned to the node.

- The "*text*" attribute represents the text that is displayed on screen. Changing the strings.xml file that can be found in the resource folder allows easy manipulation of such text. For example, the plagiarists can change the text from 'username' to 'login id' or from 'email' to 'E-mail'.

- The value of the "*resource-id*" attribute may be empty when no resource is required for that particular view. There are two ways to access a resource, in code or in XML [57]. In any case, it can be easily modified at all places where the resource-id is expected.

- The "*package*" attributes represent the package name of the app. It is common for plagiarists to modify the package name to avoid detections. Another reason to modify the package name could be that some Android markets do not allow two apps with the same package name to be hosted at the same time on their market.

- The "*content-desc*" attribute is similar to the text attribute. Modifying the strings.xml can also easily change it.

- The "*bounds*" attribute represents the position and the area of the rectangular space controlled by the views and view groups. This can also be easily manipulated by modifying the layout of the corresponding XML found in the layout folder.

In summary, to reduce the amount of computations, F1 excludes the following attributes from the birthmark: "*package*", "*index*", "*bounds*", "*text*", "*resource-id*" and "*content-desc*". The rest of the attributes represent either the state or the functionality of the view and will be addressed in the following.

Filter 2 (F2). Firstly, attributes that represent the state of the views are as follows: "*checked*", "*focused*" and "*selected*". Secondly, examples of attributes that represent the functionality are as follows: "*clickable*", "*checkable*", and "*scrollable*". The nature of certain view class is such that the value of certain attributes

in the particular class will never change. For instance, a node with `Button` class will never have the value of the password attribute equals to true. Such attributes do not provide any useful information for the app clone detection. Therefore, to further reduce the amount of computations, F2 excludes these attributes that do not provide any information gain for the generation of our birthmarks.

Consequently, the software birthmark for an app is the count vectorization of the remaining view and attribute pairs. In other words, elements in the software birthmark vector are for example, counts of clickable `Button`, counts of password type `TextView`, etc.

3.3.3 Similarity between Applications

From the previous process, we have a set of unordered birthmarks generated from multiple sets of unordered activities. In this process, we evaluate the similarities between apps based on those birthmarks. As mentioned above, the number of apps to be analyzed is large. Consequently, the intuitive pairwise comparison of all apps across the numerous app markets is almost impractical. To design an effective and efficient alternative, we are faced with two main challenges:

Challenge 1 (C1). Due to the large number of apps across multiple Android markets, we need a scalable and accurate method to compare the apps similarity.

We overcome C1 by using the LSH algorithm. LSH is a primitive algorithm frequently employed in high dimension data processing for solving approximate or exact near neighbor problems. In our approach, we cast the similarity comparison of one vector to another as a near neighbor problem. For this purpose, we use the E2LSH [58] tool. The E2LSH tool's algorithm is based on the LSH algorithm presented by Datar et al. [59]. The tool solves the following problem:

Given a set of points $P \subset \mathbb{R}^d$ *and a radius* $\rho > 0$*, for a query point* q*, find all points* $p \in P$ *with a probability of at least* $1 - \delta$ *such that* $\|q - p\|_2 \geq \rho$*, where* $\|q - p\|_2$ *is the Euclidean distance between point* q *and point* p*.*

We use the E2LSH tool to determine the near neighbor activities within a certain radius for each activity. The radius is calculated based on the Euclidean distance between the two vectors. Instead of using theoretical formulas the E2LSH tool

empirically estimates and optimizes parameters based on P. This is because theoretical formulas focus on worst-case point sets, thus less suitable for real datasets. It was mentioned in the E2LSH user manual that since the parameters are estimated, it might not be optimal in all cases. However, in our case we found that the estimated parameters provide the best result. Therefore, we keep the estimated parameters. Note that by varying the radius we can vary the trade-off between the false positive and false negative rates of our approach. Radius is a threshold that is set based on the desired acceptance of the difference between the UIs of the activities. However, it should also be balanced against the number of false positives. Based on the typical distance of corresponding activity from clones, we found radius, $\rho = 6.5$ to be a reliable value in the detection of app clones.

Challenge 2 (C2). Since the sets of activities are unordered and we do not know which activity from app A should be compared with which activity of app B. Furthermore, each activity from one app should only be compared to one other activity from the other app. To ensure that the similarities between the apps are found, we must compare the activities in a way that results in the highest similarity scores. For two apps that are similar (as in app clones), their true similarity index can only be found if the activities are correctly compared with the corresponding activities in the other app. A wrong match in the comparison of activities will result in low similarity, despite the fact that the apps are similar. On the other hand, if the apps are independently developed, their highest similarity score will still be low. Note that after matching of near neighbor (solution to C1), C2 is yet to be resolved in some cases. For example, app A has two activities, A1 and A2. App B is a clone of app A, with 2 activities as well, where B1 is similar to A1 and B2 is similar to A2. If A1 is similar to A2, then the near neighbor of A1 will possibly include B1 and B2. This is known as assignment problem, a fundamental problem in the field of optimization or operation research in mathematics. Notably, this is not a balanced assignment problem. Firstly, when the pair of apps has a different number of activities the problem becomes unbalanced. Secondly, not all activities from one app will be in the near neighbor of another app. In this case, there will be invalid assignments.

To overcome C2, we employ the Hungarian algorithm [60] that is frequently used to solve assignment problems such as finding the optimal minimum match. In our case, we employ the Hungarian algorithm to identify the optimal pairs of

most similar screens between 2 apps such that the overall Euclidean distance is minimized. The Hungarian algorithm is based on the following principle: if a constant is added to or subtracted from every element on any row or column of the cost matrix for a given assignment problem, then the optimal solution for the resulting cost matrix will have the same optimal solution as the original cost matrix. To employ the Hungarian algorithm for each pair of apps with the assignment problem, we first construct the cost matrix for the app pairs, where the cost is Euclidean distance for each activities pair between the apps. Secondly, when the total number of activities for the apps in the app pair differs, we pad the matrix dummy rows and/or columns with zero values to make the cost matrix a square matrix. Lastly, if there exist invalid assignments, such as when the pair of activities between the apps are not near neighbors, we assign these pairs with a large positive cost value.

Finally, after overcoming C1 and C2, we compute the similarity index (SI) between each pair of potential clones. The SI is computed based on the ratio between the number of similar activities matched and the maximum number of activities between the pair:

$$SI = \frac{s_A \cap s_B}{\max(s_A, s_B)} \tag{3.1}$$

In summary, to compute similarities, we first use E2LSH to find the near neighbors within a fixed radius for all activities. Then we apply the Hungarian algorithm to find the pairs of activities within the nearest neighbor that will result in the highest similarity score. Lastly, we compute similarity index based on Equation (3.1).

3.3.4 Clone Clustering

By clustering the apps into groups of clone sets we can determine how the clones are distributed across the Android markets. Furthermore, the clone sets can be used for further analysis of the relationships between the clones and to understand the behavior of the plagiarists. If the SI between two apps are above or equal to the pre-defined threshold (i.e., $SI \geq \sigma$) then they will be considered as clones. The clone set clustering is based on the following algorithm: The output is a clustering S for the apps, in which all apps in a cluster are with similarity index SI greater

than or equal to σ. Each clone set will contain at least two apps. σ can be set based on the desired number of false positive versus false negative ratio. Generally, as the value of σ increases, it would decrease the number of apps in a set and increase the probability of incurring false negatives. On the other hand, as the value of σ decreases, the number of apps in the set and the probability of incurring false positives would increase. We empirically found that the similarity threshold, $\sigma = 0.76$ is a reliable value in Android app clone detection. Figure 3.3 shows a distribution of the number of false positives and false negatives with different thresholds.

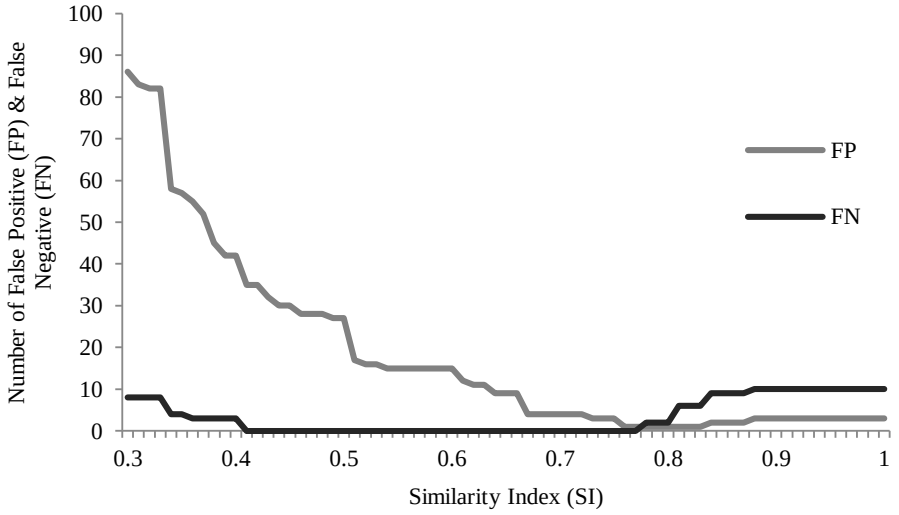


FIGURE 3.3: Number of false positive and false negative with various similarity indexes

3.4 Evaluation

We implemented a prototype of our approach in Java programming language and shell script. The experiments are conducted on Linux with 3.2GHz Intel Xeon CPU and 8 GB of RAM.

3.4.1 Dataset

In Google Play, nearly 85% of the apps are free apps (requires no cost to download) [61]. Similar trend is observed in the third-party markets, where the proportion of free apps is significantly larger compared to the paid apps. Therefore, we limit our

TABLE 3.1: Number of apps from each market

Market	**Number of apps from each market**
Google Play	105
Anruan	100
Appsapk	167
Pandaapp	149

dataset to only free apps. The dataset was collected from Google Play and three other different third-party Android markets namely, Anruan [62], Appsapk [63] and Pandaapp [64]. The apps in different markets are usually categorized differently and we believe that from the perspective of the plagiarists, popular apps have more repackaging values. Therefore, from each market we downloaded their top free popular apps. The number of apps from each market is presented in Table 3.1.

To evaluate the accuracy of our model, we would need a set of benchmark apps that are labeled accordingly. Therefore, we manually checked the set of real world apps, and labeled them as clones or unique, accordingly. With a dataset of 521 apps we have 135,460 pairs of apps. Manually comparing the 135,460 pairs of apps pairwise in terms of similarity in UI and functionality is a tedious and time consuming process. Therefore, we follow a two-phase process in which we first perform a coarse gain analysis to group apps with similar main functionality into the same category before performing fine grain analysis to identify clones within the same category. More specifically, the steps we took to label the apps are as follows:

- **Categorization.** To minimize the pairs of comparisons, we first group the apps into categories based on their main functionality. We manually installed and launched each app to briefly grasp an idea of the apps main functionality. Next, we categorize each app to a category based on what we observed. To reduce the number of apps in each category, the categories we chose are more specific. For example, for an app that has calculator as main functionality, instead of a general category such as education we assign it under the category named calculator.

- **Clones within category.** In this step, from the apps within the same category we look for clones from two aspects: 1) We manually navigated through the apps to check for similarities in the functionalities among apps.

2) Using apktool, we disassembled the *apk* to check for the similarities in their bytecode and resources. Apktool is able to decode the apps resources back to nearly its original form. If the apps are similar in both aspects, we include them in the same clone set. At the end of this process, we found 14 sets of clones with 33 apps in total.

3.4.2 False Positive

To measure the false positive rate (FPR), we execute our prototype implementation on the same dataset which we manually validated for clones. Any apps that were detected as clone by our approach, but not labeled as the clone is considered to be false positive. With radius $\rho = 6.5$ and threshold $\sigma = 0.76$, we found 15 sets of clones with 35 apps in total. Out of these 15 sets of clones, a clone set of 2 apps is found to be false positive, the remaining 14 sets were correctly detected. Therefore, with radius $\rho = 6.5$ and threshold $\sigma = 0.76$, our false positive rate, FPR $= 0.4\%$.

We further examine the two false positive apps and found that both apps consist of only two activities each and each activity only has a single view element. The number of clone apps found from each market is shown in Table 3.2. Figure 3.4 shows the distribution of the similarity index among the apps pairs.

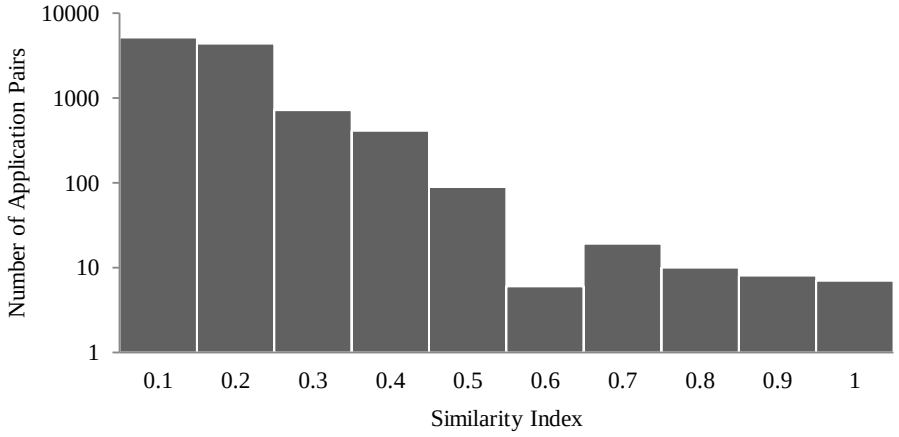


FIGURE 3.4: Histogram of detected application pairs similarity index

TABLE 3.2: Number of apps detected as clone

Android App Markets	**App counts**	**% relative to market (%)**
Google Play	8	7.6
Anruan	8	8
Appsapk	7	4.8
Pandaapp	12	7.1

3.4.3 False Negative

Any apps that are labeled as clones from our manual inspection, but not detected by our approach are considered to be false negatives. As aforementioned, our approach detected all the clone sets that were manually validated. Since our approach detected all app clones, our approach does not have any false negative. Therefore, with radius $\rho = 6.5$ and threshold $\sigma = 0.76$, our false negative rate, $FNR = 0.0\%$.

3.4.4 Efficiency

We evaluate the efficiency of our approach from two aspects. The first aspect concerns the efficiency of our approach to perform feature extraction, which is the time needed to obtain the UI hierarchy dumps from *apk*s. The second aspect concerns the efficiency of our algorithm, which is the time needed to detect clone sets given the XML of UI hierarchy dumps.

The amount of time necessary for the extraction of the XML is proportional to the number of activities in the app. For the dataset of 521 apps there are a total of approximately 16,000 activities. Figure 3.5 presents the relationship between the number of activities and the time taken to obtain the XML from a single Android emulator. In Figure 3.5, we can see that the time needed increase linearly with increasing number of activities. On average it takes less than 2 seconds per activity to dump its UI hierarchy. Figure 3.6 presents a histogram of the number of activity components within the apps in logarithmic scale. For better presentation we split the activity counts into bins of 50. In Figure 3.6, it is clear that a majority of the apps contain less than 50 activities.

After applying the filter F1, we are left with 10 attributes from each node in the XML for birthmark generation. After applying the Filter F2, the number of elements in the birthmark vectors is reduced by nearly 40%. For the dataset of

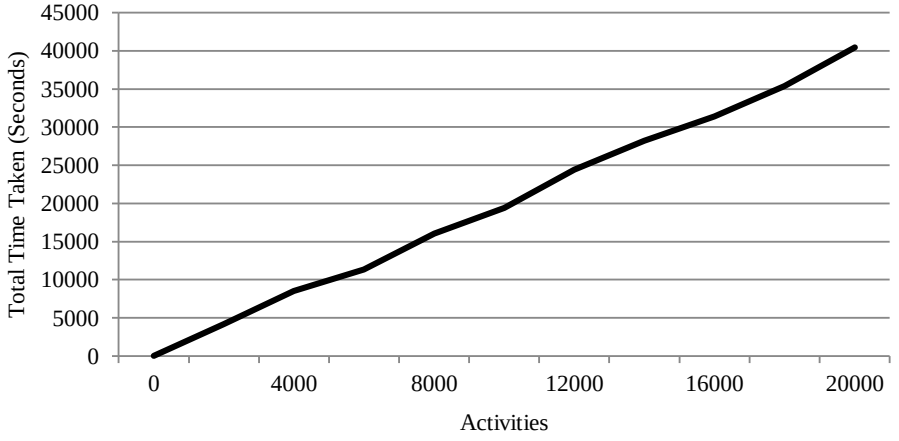


FIGURE 3.5: Time taken to collect XMLs

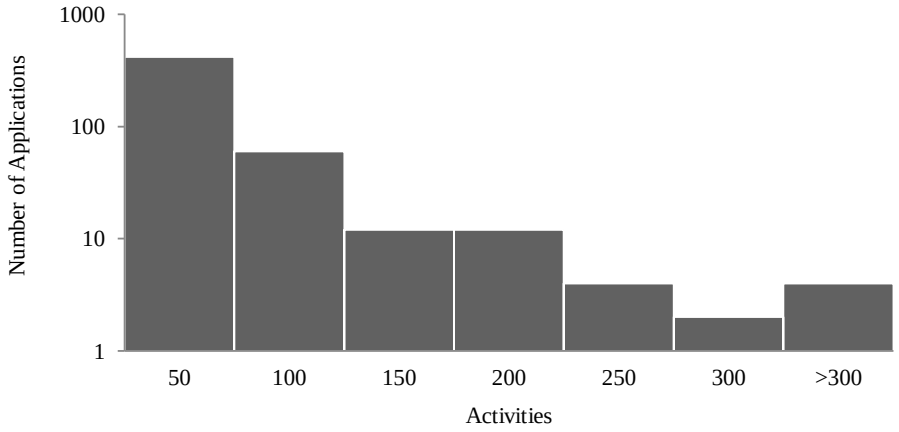


FIGURE 3.6: Histogram of activities counts per application

521 apps, the time from birthmarks generation to clone sets output takes about 55 seconds.

3.4.5 Comparison with Existing Approach

In this section, we evaluate our prototype implementation on a dataset of clone samples provided by a research group [19]. The dataset includes 259 apps which are divided into 99 clone sets as detected by their algorithm, with each clone set consisting of two or more apps. Based on this dataset, we evaluate the ability of our approach in detecting different types of clones. In addition, we also measured the false negative rate of our approach on this dataset.

One of their clone sets contains three service apps. These non-typical apps are not within our scope because unlike typical Android apps, these apps execute services

in the background and do not require UI for user interactions. With radius $\rho = 6.5$ and threshold $\sigma = 0.76$, we detected 96 clone sets and some of which are not identical to clone sets they have detected. We manually investigated all cases where our clone sets do not coincide with their clone sets.

Firstly, apps in two of their clone sets were divided into four different clone sets by our approach. When we check the apps in these four clone sets, we found that the apps within each clone set were indeed more similar. There were some similarities between these apps, but we do not perceive them as clones as their functionalities are different. Secondly, we found three sets of clones that were falsely included in their clone sets. The apps in each of these three sets were signed by the same developer key but with different functionalities. We believe that for these apps, the high similarity scores resulting from code base detection are due to developers reusing most of their codes in their other apps. Lastly, another mismatch we have is because both apps in the set consist of only a few activities and some activities are not similar. The differences in activities between these two apps are due to different third party libraries. The SI between these two apps is 66.6%. This shows that our approach is able to detect their similar activities. For this dataset, with radius $\rho = 6.5$ and threshold $\sigma = 0.76$, our false negative rate, FNR = 0.8%.

3.5 Discussion

3.5.1 Accuracy and Efficiency

In Section 2.5, we have discussed three categories of repackaging attacks: lazy attack, amateur attack and malware attack. From the above experiments, we further analyze the types of clones detected and evaluate the ability of our approach to detect each category of the repacking attack. Of all the clone sets we detected, we found 14 clone sets with at least a pair of apps belonging to amateur attacks. For example, we found clones with different social media functions which partially affects the functionalities and UIs of the app. In addition, we also found some apps that are translated and signed with a different developer key. Further investigation is required to determine whether it is an actual clone case or a legal translation. All other apps in the remaining clone sets belong to lazy attacks.

To determine whether the apps are malware, we upload them to VirusTotal [65] for scanning. VirusTotal is a free online service that allows user to upload a suspicious file for analyzing to identify malware. About 67% of the clones detected from both experiments are reported as malware by at least one of the scanners from VirusTotal. Most of the malware are adware, Trojan horses and there are also a number of spyware. Note that malware attack can, at the same time, be classified as lazy attack or amateur attack. In other words, a lazy attack or an amateur attack with malicious payload attached will also be classified as a malware attack. In our detected clone sets, most of the malware attacks are primarily classified as lazy attacks.

In the following, we discuss the effectiveness of our approach to detect the attack in each of the repacking categories:

Lazy attack. In lazy attack the plagiarist makes only simple changes to the apps. Such an attack has little or no impact on the views in the activities. Therefore, our approach can effectively detect lazy attacks.

Amateur attack. An amateur attack requires more effort and knowledge from the plagiarist. They employ automatic code obfuscation and also make small changes to the functionality of the app. Code obfuscation will not affect the UIs of the activities. The small changes in the functionality of the app may or may not affect the UIs in some of the activities. Our approach detects similar activities and not identical activities. In other words, it can tolerate a certain degree of changes in the view hierarchy. Moreover, the small changes in the functionality may affect only some activities but not all activities. If there are many activities that are not affected by the changes, it will not affect our overall detection.

Malware. The clone with malicious payload attached often masquerade as the original app by keeping the functionality and user interface similar to the original app to leverage on its popularity. Since our approach focuses on detecting similar UI, we can effectively detect this type of attack. Furthermore, the additional malicious payload usually does not include an additional activity that has views visible to the user. Thus, it will not affect our detection.

In summary, our approach is effective in detecting lazy, amateur and malware attacks. However, our approach may not be able to detect certain amateur attacks,

depending on the extent of the changes to the apps. Nevertheless, it is not common for the plagiarist to bother understanding its bytecodes to make extensive modifications to the app. We note that there is no general solution for detecting similar app with extensive changes. In addition, our approach certainly raises the bar for the plagiarist to repackage apps without being detected when uploaded. To avoid detection by our approach by modifying the UIs without proper planning will affect the user experience and in turn affect the popularity of their repackaged apps. Proper modification of the UI requires the plagiarists to put in extra effort to redesign the UI carefully.

The performance bottleneck of our approach is the speed of the emulator that limits the efficiency of dumping the UI hierarchies from *apk*s. However, this process can be parallelized for better efficiency. For example, we can use multiple computers with each computer running multiple emulators.

3.5.2 Limitations and Future Work

Firstly, given that our approach is based on runtime UI information, it is obvious that our approach is unable to detect clones in service apps since they just provide services in the background and usually do not contain UIs that are necessary for user interaction.

Secondly, based on the results obtained from the evaluation on both the datasets, we observed that apps with a small number of activities or with low view counts in the activities may increase the false positive rate. However, we noted that this is a common limitation for most birthmark or finger print clone detection approaches.

Thirdly, our approach may also be limited by apps with multiple data dependent activities. For example, apps that require login credential is one of them, this category of apps usually restricts their entry point to the main activity with login UI. Any attempt to enter into the app from other activity will fail or be redirected to the main login activity. The information that can be obtained from these apps is limited, thus affecting the accuracy of our approach.

Lastly, our evaluation is based only on free apps and the result may be different for paid apps, which requires the users to pay certain fees before the download is available to them. Cloning may be more prevail in paid apps, since users would be

more likely to choose the similar but free apps. Conversely, the plagiarists may be less willing to purchase the app to repackage it.

For future work, we would like to conduct more in-depth studies on how different types of repackaging attacks can affect the UIs. We would also like to explore additional alternatives to extract more relevant information from the *apk*s, such as those data dependent activities. These would help us to design better software birthmarks that may further improve our accuracy and efficiency. Due to the complex nature Android apps, code or resource analysis alone is insufficient for clone detection. We believe that a hybrid approach with both static and dynamic analysis, which integrates both code and UI information for app clone detection, would be a very promising and interesting research direction.

3.6 Related Work

A number of previous studies have already been conducted on Android clone detection. Most of the existing approaches only focus on code-based similarities.

Zhou et al. [30] proposed an app similarity measure system, DroidMOSS, to detect repackaged apps in third-party Android markets. DroidMOSS first compute the fuzzy hashes for each method within the app to generate a fingerprint for the app. They then compute a similarity score based on the edit distance between the 2 fingerprints.

DNADroid [51] first detect potential similar apps based on their meta information that is used to describe the app. In the second stage, DNADroid creates the program dependency graph (PDG) as a fingerprint for each app that is to be compared. Lastly, they apply a filter to prune unlikely clones, before comparing the rest of the PDG pairs that passed the filter using a subgraph isomorphism.

Hanna et al. [53] proposed Juxtapp, a tool to detect code reuse among Android apps. Juxtapp uses k-grams of the opcode sequences and apply the feature hashing to extract the feature of the apps. Juxtapp can identify vulnerable code reuse, instance of known malware and pirated copies of the original apps.

Androguard [66] provides a similarity measure tool that supports several standard similarity metrics. The similarity is computed by comparing similar methods in the

dex code of the apps. Rather than detecting cross-market app clones, Androguard is meant for finding the difference between 2 apps on a small set of data.

Zhou et al. [42] focused on the problem of detecting piggybacked' apps, which are clones with additional malicious payload attached. They first perform module-decoupling technique to split the code into primary and non-primary modules. They then extract a semantic feature fingerprint for each primary module and use a linearithmic search algorithm to detect similar apps.

Chen et al. [19] extracted the methods from the apps and construct a 3D-control flow graph (3D-CFG) to get the centroid. They then leverage the centroid to measure method level-similarity across multiple markets. Lastly, the method-level similarity result is used to group similar apps together.

Wang et al. [39] proposed a two-phase approach to perform Android app clone detection, where the first coarse-grained detection phase compares light-weight static features to identify potential clones and the second fine-grained phase compares more detailed features for those apps identified in phase one. They also proposed a clustering-based approach to first filter out the third-party libraries from the apps, to improve the detection accuracy and efficiency.

Linares-Vasquez et al. [38] proposed CLANdroid which leverages on advanced information retrieval techniques with five semantic anchors such as APIs, sensors, permissions, intents and identifiers to detect similar apps. They have also performed a study on the impact of third-party libraries and their results suggest that excluding third-party libraries has significant effect on the detection accuracy.

Glanz et al. [67] detect repackaged apps via a two step approach which first identify and remove third-party libraries code in the apps, then fuzzy hash the abstract of the remaining app code.

All these approaches focus on static code-based detections that are vulnerable to advance obfuscation techniques. On the other hand, there are a few existing work that detect Android app clones without relying on code similarities.

Differing from code similarity based approaches, FSquaDRA [68] detects Android app clones based on the comparison of the resource files that are necessary for creating the *apk*. They leverage on the hashes that were computed and stored in

the package during the process of app signing. This approach is resilient to code obfuscation, but some small changes in the resources will affect the similarity.

Viewdroid [34] proposed a user interface based approach to detect app repackaging. Similarly, MassVet [26] performs UI analysis to identify apps with similar view structure then further analyze them to identify malicious payload. Our approach is similar to both Viewdroid and MassVet in the sense that both our approaches leverage on UI information. However, they construct view graph based on static analysis of the control flow relationship between the views within the app. Our approach differs from them in that our birthmark information is collected from runtime and we generate vectors from this information.

3.7 Conclusion

In this chapter, we presented a novel approach to detect Android app clones based on birthmarks generated from runtime UI information. Our approach uses locality sensitive hashing to find a near neighbor for similar birthmarks and apply the Hungarian algorithm to find the optimal activities pairs with the overall highest similarity. The result shows that our approach can effectively detect different types of repackaging attacks such as lazy, amateur and malware attacks with low false positive (FPR=0.4%) and false negative (FNR=0.8%) rates. Many of our detected apps (67%) belong to malware attacks, this call for a more rigorous vetting across all Android markets. This work helps the reader to understand the UI of Android apps and how this information can be applied to clone detection. We believe that our study supports a new research direction or can be used to complement existing code-based approaches in mobile app clone detection.

Chapter 4

LibSift: Automated Detection of Third-Party Libraries in Android Applications

4.1 Introduction

The flexibility of Android OS allows the developers to easily incorporate TPLs into their apps to ease the development process. Consequently, a particular characteristic of Android apps is that they typically compose of several TPLs. The usage of TPLs escalates the problem of Android security further, as the use of TPLs typically causes the app to include a significant amount of code that is irrelevant and may hinder the process of many program analysis tasks. Wang et al. [39] reported that TPLs generally contribute to more than 60% of the app's code.

There are generally three prominent areas of program analysis tasks that are affected by TPLs. Firstly, in app clone detection the similarities of the code between the apps need to be measured. Since TPLs can be easily manipulated, not considering all the TPLs for the analysis may greatly affect the results. A recent study [38] has shown that the computation of code clone similarities in Android app is significantly affected by the consideration of including or excluding TPLs. Secondly, static taint analysis is a time consuming and computational intensive process. However, a significant portion of the resources are spent on analysing irrelevant code when the TPLs dominates the app. Researchers have reported that

they have encountered these problems. For instance, DroidSafe [40] reported that while using FlowDroid [69], a state-of-the-art static taint analysis tool, the analysis of some apps took more than two hours and AsDroid [41] reported that FlowDroid ran out of memory on some apps. Since TPLs are typically used as it is without any modification, the TPLs can be separately analysed and have the results be reused in the analysis of the apps that use them [70]. Lastly, in malware detection, the presence of TPLs may dilute the features used in the detection, thus reducing the contrast between benign and malicious samples and affecting the results of the detection. For example, advertisement libraries are excluded from analysis in MUDFLOW [71] as advertisement libraries are frequently used in Android apps and their dataflows become common and dilute the training data. Furthermore, DroidAPIMiner [72] reported that filtering out TPLs increases the difference of API usage between malware and benign apps.

Apart from hindering many program analysis tasks, the usage of TPLs is also associated with more privacy risks and security threats [35, 73]. For example, popular TPLs may be masqueraded by malicious libraries to mislead users into thinking that it is a legitimate TPL. Furthermore, some TPLs, even the popular ones, are known to be aggressively collecting the users' private data. By identifying the TPLs in the apps, we can effectively address these library-centric threats. Grace et al. [74] reported that most of the existing advertisement libraries collect private data and some run code fetched from the Internet. In another recent study, the authors have demonstrated that TPLs are likely a significant medium for the propagation of malicious code [75].

Consequently, several existing literatures have made effort to identify and exclude TPLs from various Android app analytic tasks [19, 30, 51, 72]. One of the most commonly used technique for detecting TPLs in Android apps is to match the names of the packages in the app to the TPLs package name listed in the whitelist. However, due to the widespread interest and the dynamic nature of Android ecosystem, it is difficult to build a comprehensive whitelist of TPLs as there are too many of them. Furthermore, it is common for Android apps to employ obfuscation techniques that may, for instance, rename the packages, classes and methods. For example, a non-obfuscated app with a package *com.library.example*, after obfuscation, the package may become *com.a.a*. Thus, this common obfuscation technique will cause the whitelist approach to fail in detecting the obfuscated TPL.

Despite the importance of identifying TPLs in Android apps, there are only a few existing studies focusing on it. LibRadar [2] uses as feature the frequency of different Android APIs in each pacakge, and performs feature hashing on a large number of apps. Following that, strict comparison is enforced to perform clustering and identify TPLs that are frequently used. Li et al. [3] put together a large whitelist of popular TPLs by refining a list of package names that have frequent appearance in a large number of apps.

In this chapter, we propose `LibSift`, an approach to identify TPLs in Android apps based on the package dependency graph (P_kDG) of the app. We evaluate the effectiveness and efficiency of `LibSift` on a real-world dataset of 300 Android apps. We further compare `LibSift` with two state-of-the-art TPLs detection approaches, LibRadar [2] and the whitelist from Li et al. [3].

Our proposed approach is inspired by PiggyApp [42] that is the most related study to our work, but it addressed a different problem. In PiggyApp, its goal was to identify piggybacked apps[1] that share the same primary module. Their main focus was on identifying the primary modules and pay no attention to the other modules. On the other hand, our approach focuses on the non-primary modules for detecting TPLs in Android apps. The intuition behind our technique is that typically the code of every Android app can be divided into primary module and non-primary modules (if any) by using module decoupling technique. The primary module defines the app's core functionalities and the non-primary modules are contributed by TPLs. The intuition comes from the programming paradigm that the code within each module, primary or non-primary, is tightly coupled, whereas, the code between modules is loosely coupled or even standalone. Furthermore, dependency graph techniques have been proven to be resilient to multiple types of common obfuscation techniques, such as renaming, statement manipulation and program transformation [51]. Another advantage of our approach is that, on the contrary to the state-of-the-art TPLs detection approaches, our approach does not assume that all TPLs will be used by many apps. Thus, we are able to detect even the non-popular TPLs that do not appear in many apps. In fact, in our evaluation, we show that `LibSift` can effectively detect the less popular TPLs missed by both of the state-of-the-art approaches. In addition, for our approach, it is straightforward to update the list of TPLs when new library versions or new TPLs are detected.

[1] Piggybacked apps refers to a type of repackaged app which involves the injection of rider code into the original app.

Contributions. Our main contributions in this chapter are as follows:

- We propose a novel approach to automatically detect all TPLs used in Android apps based on package dependency graph and without first having to study a large number of apps.
- We implement a prototype of our approach, `LibSift`, and conduct extensive experiments to evaluate the effectiveness and efficiency of our approach to detect TPLs in Android apps.
- We compare our approach with two state-of-the-art approaches, LibRadar [2] and whitelist from Li et al. [3], and show that our approach can detect libraries that are not detected by them.

4.2 Proposed Third-party Library Detection

4.2.1 Overview

In the software context, the programming paradigm is such that a library is generally coded in a modular fashion, such that the code within the library is tightly coupled and highly cohesive. Coupling refers to the interdependencies between modules, while cohesion describes the degree of relationship between the functions that are within a single module. Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large. TPLs written for Android apps follow the same principle. Android apps are generally written in Java programming language and thus posses Java's property of organizing code into packages. As a result, Android apps can be decoupled into individual modules, where the code within each module is tightly interwoven and the code between each module is independent or loosely coupled. Based on the above-mentioned observation, we propose to detect all TPLs used in a given Android app by making use of the natural partitioning of the Android apps and perform module decoupling at the package level.

In our approach, we use a module decoupling technique similar to the technique that was introduced in PiggyApp [42]. PiggyApp performs module decoupling to identify the primary module of the apps and uses it to detect Piggybacked apps.

Despite addressing different issues, since our technique is similar in nature, it is notable to highlight the differences between our techniques. Firstly, package homogeneity[2] is considered as a dependency relationship with high importance in PiggyApp. Whereas in `LibSift`, we do not consider package homogeneity as a dependency, instead we view them as a criteria to merge the packages into a module. Secondly, we perform module decoupling using different algorithms. PiggyApp performs agglomerative clustering to cluster the packages with dependencies weight greater than a pre-defined threshold, starting from the most tightly coupled pair and stop when there are no clusters with dependencies weight greater than threshold left. However, `LibSift` checks for package homogeneity before merging the packages and if more than one sibling package belong to a module, all sibling packages sharing the same parent will be assign to the same module. Finally, the method we use to identify the primary module also differs. In PiggyApp, the primary module is identified as the module that provides the main activity or the module that handles the most activities of the app. However, `LibSift` identifies the primary module based on the app's package name or as the module that have dependencies with most number of other modules.

Figure 4.1 shows the overall architecture of our approach to detect TPLs in Android apps. `LibSift` consists of four main processes, disassembling, constructing P_kDG, module decoupling, and identifying primary module. The disassembly process is necessary to reveal the information required for the next process. The construction of the P_kDG involves the analysis of the bytecode information to extract dependencies between different packages. For module decoupling, we cluster the packages of the given app into separate modules based on its P_kDG. Lastly, if the app contains more than one module, we identify the primary module from the set of modules.

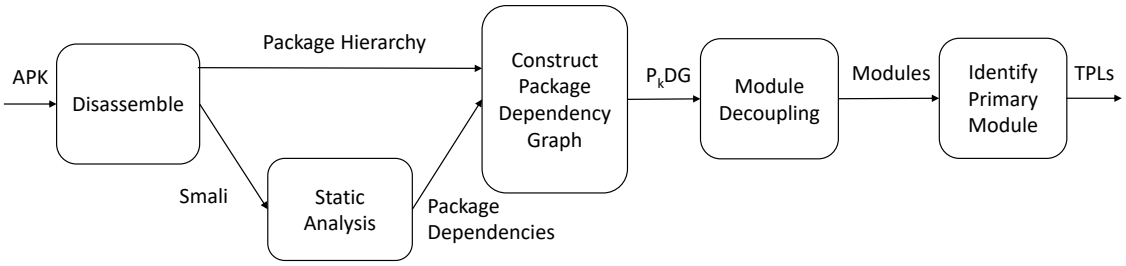


FIGURE 4.1: Overview of `LibSift`

[2]We consider two packages as homogeneous if they form a parent-child or sibling relationship

4.2.2 Disassemble

The Android OS uses *apk* for distribution and installation of apps. When the app developer compiles and builds the app, all the app code are compiled into a single dex file, thus, losing the clear separation between the TPLs and the core app code. Following that, the dex file along with the necessary resources are packaged into an *apk*. Fortunately, the package hierarchical information of the app's code is preserved during the compilation process and we can use it along with package dependency information for TPL detection.

In order to do so, given an *apk*, we first disassemble it using apktool [47] to reverse engineer the classes.dex file to smali files, which also reveals the package hierarchy information within the app. During the disassembly process, the bytecode in the dex file is converted into smali [33] intermediate representation in the form of multiple smali files which is analogous to the class files of the Java programming language. Furthermore, these smali files will be placed in their respective folder based on their package hierarchy. As such, we can now analyze the smali files and extract the package dependencies information.

4.2.3 Package Dependency Graph

The P_kDG shows the degree of dependencies (undirected) between the packages of an app. Note that we do not consider control dependencies when building the P_kDG, instead we focus on dependency relationships, such as class inheritance, method calls, and member field references. Leveraging on both package hierarchy information and dependency relationships, we construct a $P_kDG = (N, E, W)$. Where N is a set of nodes and each node $n \in N$ represents a package in the app (note that we only consider packages with smali files directly under them). $E \subseteq (N \times N)$ is a set of edges and each edge $e(n1, n2) \in E$, where $n1, n2 \in N$, connecting any two nodes represents that there is a dependency between these two nodes. Lastly, W is a set of labels representing the weights of the edges and each weight $w \in W$ is the degree to which the nodes connected by the edge $e(n1, n2)$ are dependent on each other. As different relationships represent a different degree of dependencies, we should assign them with different weights. Except for package homogeneity, we use the weight assignments suggested in PiggyApp, which are 10 for class inheritance, 2 for method calls, and lastly, 1 for member field references.

An example of the P_kDG for a Korean language learning app is as shown in Figure 4.2. In this particular app, there are 15 sub-packages that contain smali files directly under them. Some of the sub-packages that do not have dependencies with any other sub-package are depicted as nodes without edge.

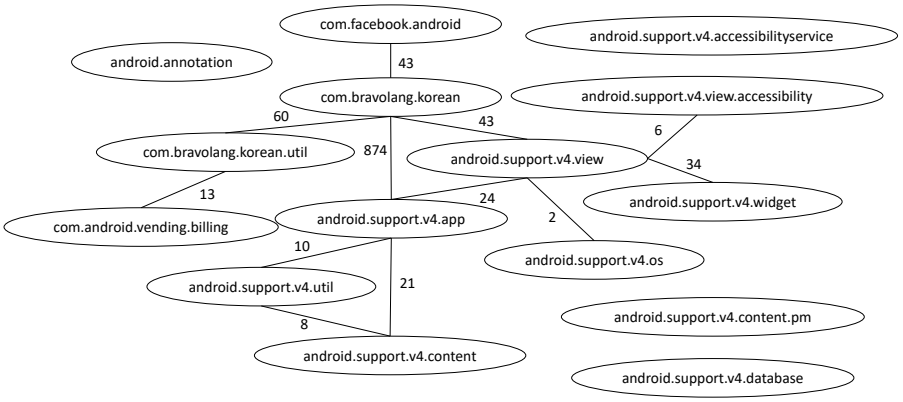


FIGURE 4.2: Package dependency graph example

4.2.4 Module Decoupling

We perform module decoupling based on package level semantic information. Each module consists of at least one package and every package in a particular module has a parent-to-child or sibling relationship to at least one other package in said module. However, we argue that sharing the same parent in the package hierarchy does not necessarily means that they are related. For example, the following two popular libraries, *com.google.ads* and *com.google.analytics*, they share the same parent *com.google* and may be both considered as libraries from *Google*, but in actual fact, they are two separate libraries that can function independent of each other. Hence, they should be reported as separated libraries.

Given the above observations, we analyze the P_kDG to cluster the packages into modules based on the following conditions: 1) Package homogeneity 2) Total weight between the two packages is greater than the pre-defined threshold and 3) If package $p1$ and package $p2$ are sibling nodes from the same module then all the other sibling nodes of $p1$ and $p2$ are from the same module. As mentioned above, simply sharing the same parent module does not means that they are related. However, if there are high dependency between some sibling nodes, then it is likely that all other sibling nodes belongs to the same module. Our module decoupling algorithm is presented in Algorithm 1.

Algorithm 1: Module Decoupling Algorithm

Input: P_kDG - Package Dependency graph of the app
threshold - Pre-defined threshold
Output: M - Set of Modules of the app

```
foreach edge e in P_kDG do
    if weight w > threshold then
        (n1, n2) ← get_nodes(edge)
        if package_homogeneity(n1, n2) == TRUE then
            M ← merge(n1, n2)
foreach node in the P_kDG do
    if sibling_nodes in a module then
        M ← merge(node, M)
return M
```

The P_kDG of the same app in Figure 4.2 after the module decoupling process is shown in Figure 4.3. As evident in Figure 4.3, the 15 sub-packages of the app are successfully merged into 5 modules. The sibling nodes such as *android.support.v4.view* and *android.support.v4.widget* are merged into the parent node *android.support.v4*. Furthermore, since *android.support.v4* is already declared as a module, all child nodes under *android.support.v4* in the package hierarchy are merged into this module. As the packages can only be merged when package homogeneity is true, the TPLs *com.facebook.android* and *android.support.v4* are not merged with the primary module *com.bravolang.korean*.

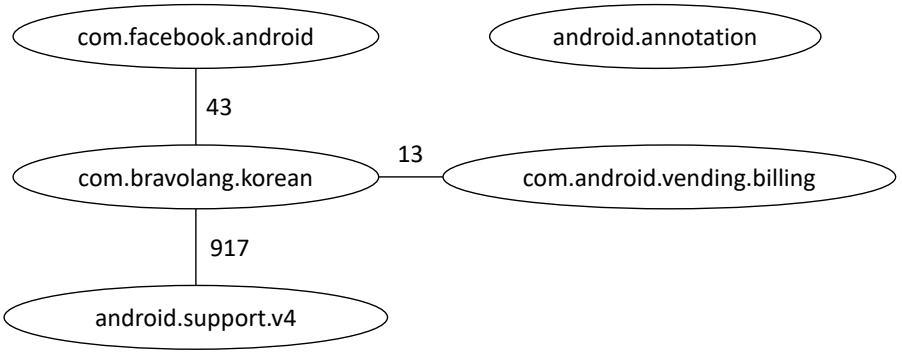


FIGURE 4.3: Package dependency graph after module decoupling

4.2.5 Identify Primary Module

The primary module provides the primary function of the app and the core app code resides in this module. For certain Android app security analysis, especially Android app clone detections, it is important to identify the primary module for similarity analysis as the other modules are TPLs that can be easily replaced with other TPLs of similar functions.

We observe that frequently, the primary module can be simply inferred from the app's package name or is a subset of the app's package name. Note that app's package name refers to the unique package name declared in the AndroidManifest.xml that uniquely identifies the app on the device and not to be confused with package name which refers to the name of the packages in the app. Using the same Korean language learning app as an example, its app's package name is *com.bravolang.korean*, and the core app code written by the developer is all enclosed within the *com.bravolang.korean* sub-package, therefore the primary module can be correctly identified as *com.bravolang.korean*. This common practice is due to the fact that Android app development following the Java programming language package naming convention, that suggest that developers use their reversed Internet domain name to begin their package name to avoid conflict with other apps. Moreover, using each period in the package name as a path separator, all the code by the same developer would be placed together in the path hierarchy.

However, in some cases, we are unable to infer the primary module just from the app's package name. One of the main causes for this is the use of obfuscation which renamed the packages. In these cases, we identify the primary module base on the reasoning that the primary module contains the core code of the apps that is in charge of communicating with the TPLs to access their resources. Furthermore, TPLs are designed to work independently and are not dependent on other TPLs to function, this means that non-primary module should not have dependencies on other non-primary modules. Therefore, the primary module can be identified based on the highest number of other modules it has dependencies on. Based on the P_kDG, for each module, we can identify its dependencies on other modules and compute the total number of other modules it has dependencies on.

In summary, to identify the primary module, we first use the app's package name and match it with the modules that we have identified. If a match is found, the

matching module will be the primary module. However, when we fail to identify the primary module through matching the package name, we will identify the module that communicates with the most number of other modules as the primary module. The algorithm we use to identify the primary module is shown in Algorithm 2.

Algorithm 2: Identify Primary Module

Input: M - Set of Modules of the app app_package_name - unique package name of the app

Output: primary_module

```
1 foreach module m in M do
2     if match_pacakgeName(module, app_package_name) == TRUE then
3         return module
4 return highest_dependent_count(M)
```

4.3 Evaluation

We have implemented a prototype of `LibSift` in Python code to perform preparations, module decoupling and identification of primary module for the detection of TPLs in Android apps. In the preparation step, we first disassemble the *apk* of the given app, using apktool [47]. Our python script then automatically extract the packages from the app. Following that, we parse the smali code to identify the dependencies between the packages and build a weighted P_kDG. Finally, we cluster the packages into modules based on Algorithm 1 and identify a primary module using Algorithm 2. We evaluate the accuracy and efficiency of `LibSift` on a set of real-world Android apps consisting of 300 real-world apps collected from the top popular apps from different categories in Google Play store. In addition, we also compare `LibSift` with two state-of-the-art TPLs detection approaches. All evaluation experiments are performed on a Linux machine with 2.6 GHz Intel core CPU and 32 GB of RAM.

4.3.1 Module Decoupling and Validation

Based on our experiments, we empirically determined that a cut-off threshold = 15 is a suitable value to accurately decouple the modules in Android apps. A high threshold is more likely to fail to identify and group related packages belonging to

TABLE 4.1: Module decoupling results summary

Description	Number of modules
Total number of modules	5,460
Number of obfuscated modules	802
Minimum number of modules in an app	1
Maximum number of modules in an app	76
Average number of modules in an app	18.2
Standard deviation of modules in an app	13.77

the same module that do not have high dependencies. On the other hand, a low threshold is more prone to falsely group unrelated packages with low dependencies into the same module.

A summary of our module decoupling results with a predefined threshold of 15 on a 300 apps dataset is presented in Table 4.1. The total number of modules detected by `LibSift` in all 300 apps is 5,460. The breakdown of the numbers of obfuscated and non-obfuscated modules detected by `LibSift` are 802 and 4,658 respectively. The minimum, maximum and average number of modules per app, detected by `LibSift` are 1, 76 and 18.2, respectively. The standard deviation for the number of modules per app is 13.77. This shows that most Android apps do indeed use multiple TPLs.

To evaluate the module decoupling accuracy of `LibSift`, we manually analyse the 300 apps and verify that the modules are correctly decoupled. The results show that our approach correctly decoupled 296 apps. Therefore, the accuracy of `LibSift` is 98.67%. In the rare occasions where the apps are incorrectly decoupled, we found out that it is due to the packages in the primary module not having any dependencies on each other. This causes the primary module to be separated into multiple modules and thus, falsely identified as TPLs, when they should be part of the primary module.

4.3.2 Primary Module

To evaluate the accuracy of `LibSift` in identifying the primary module, we manually verify the primary module of the 296 apps that are decoupled correctly. Note that if there are more than one potential primary module, `LibSift` reports all of them. However, in such cases, since `LibSift` is unable to pin point the primary

TABLE 4.2: Primary module identification summary

Description	Number of apps
Identified from app's package name	273
Unable to to identify from app's package name	23
Based on dependencies with number of other modules	
Only 1 module with highest count	16
Primary module identified correctly	10
More than 1 module with highest count	7
One of the module with highest count is primary	3

module, despite having the correct primary module among the list of potential primary modules, we consider it as `LibSift` fails to identify the primary module for those apps with multiple potential primary modules reported. A summary of the the primary module identification statistics is presented in Table 4.2. The results show that 283 apps have their primary modules identified correctly. Therefore, the accuracy of `LibSift`'s primary module identification is 95.61%. The reason `LibSift` fails to identify the primary module is two-fold. Firstly, the name of the packages for the primary module could be obfuscated or different from the package name and therefore, cannot be identified by simply matching them. Secondly, in some rare cases, the TPL represents the core of the app and is used to provide communications between different TPLs.

Out of the 296 apps that are decoupled correctly, the primary module of 23 apps cannot be identified from its package name. Therefore, `LibSift` attempts to identify the potential primary module by looking for the module that has dependencies with the most number of other modules. For 16 apps out of the 23 apps, there is only 1 module in each app that has dependencies with most other modules. Among these 16 apps, 10 of them have their primary module identified correctly. Whereas for the other 7 apps out of the 23 apps, more than one potential primary modules are identified in each app as there are multiple candidates with dependencies on an equal number of modules. Out of these 7 apps, 3 of them contain the true primary modules.

4.3.3 Performance

After dissembling the *apk*s, the total time taken for `LibSift` to detect TPLs in all 300 apps is less than 27 minutes. Figure 4.4 shows the breakdown of time taken to

detect TPLs in each app, sorted in ascending order. Only 1 app took more than 1 minute to process and about 90% of the apps took around 10 seconds or less. Notably, these are achieved without prior studies of a large number of apps. To further improve the efficiency of `LibSift`, since `LibSift` does not require to cluster a large number of apps, it can be easily parallelized by distributing the workload to multiple machines.

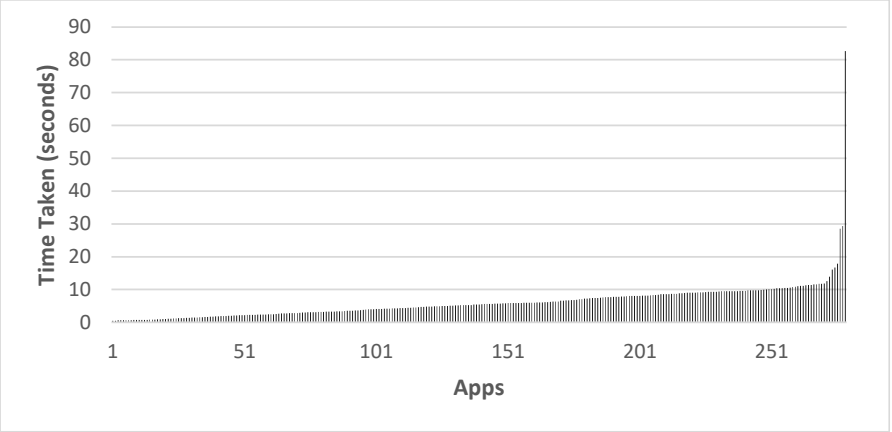


FIGURE 4.4: Time taken by `LibSift` to process each app

4.3.4 `LibSift` vs. LibRadar and Whitelist

In this section, we compare our approach with two state-of-the-art approaches. One of them is LibRadar [2], a TPLs detection tool for Android apps which is based on API features of packages in the apps. It uses a clustering based approach to cluster the hashes of the packages in a large dataset of apps. The packages that are clustered into large clusters, greater than a pre-defined threshold, are identified as TPLs. Another approach is by far the largest set of TPLs whitelist collected by Li et al. [3]. The list of TPLs is harvested from a large dataset of Android apps by extracting the names of the packages and clustering them based on the frequency of occurrence and followed by a series of refinements.

We perform TPLs detection with `LibSift`, LibRadar, and whitelist from Li et al. [3] on the same set of 300 real-world Android apps and the results are presented in Figure 4.5. The Y-axis represents the total number of libraries detected by each approach for the corresponding apps along the X-axis. As shown in the Figure 4.5, `LibSift` is capable of detecting more TPLs in most cases. For the 300 apps, the

average numbers of TPLs detected by `LibSift`, LibRadar and whitelist from Li et al. are 17.17, 7.68 and 7.93, respectively.

In general, the TPLs not detected by LibRadar are the less popular ones. In addition, we also observed that in some apps the popular TPLs are detected by LibRadar, but the same TPLs are not detected in some other apps. Upon further investigation, we found out that it is due to different versions of the TPLs using slightly different APIs. As mentioned in their paper, LibRadar [2] enforces strict comparison, such that, two packages can only be cluster together when they share the exact same features (APIs). This results in the detection of multiple versions of the same TPL. However, in the case where the particular version of the TPL is not used frequent enough, possibly due to reason such as short update intervals, despite being a popular TPLs, this version of the TPL will not be detected. On the other hand, the TPLs not detected by using the whitelist from Li et al. are generally the less popular TPLs and those that have their package names obfuscated. Furthermore, Li et al. [3] stated in their paper that they do not list libraries starting with the name *"android.support"*. It is worth noting that when using the whitelist, unlike semantic based approach, all versions of the popular TPLs will always be detected as long as the package name remains the same and non-obfuscated.

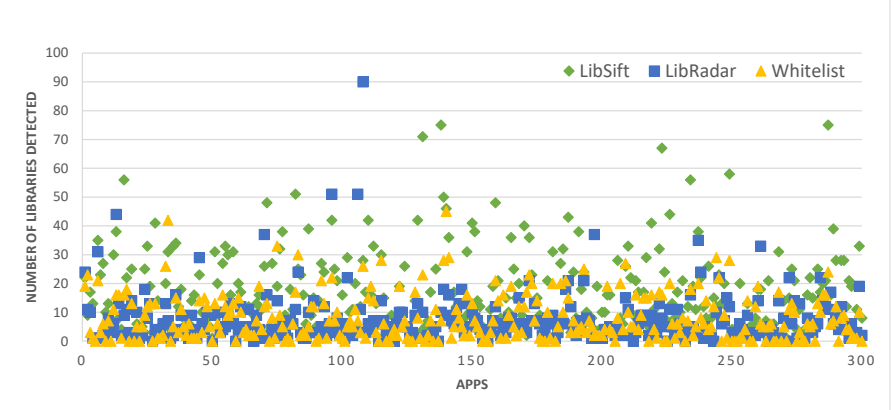


FIGURE 4.5: Number of TPLs detected by `LibSift`, LibRadar [2] and Whitelist [3] for 300 apps

With the *apk*s disassembled, the total time taken for each approach, `LibSift`, LibRadar and whitelist to detect TPLs from the 300 apps are 1,617, 1,875, and 76 seconds, respectively. Unsurprisingly, the usage of whitelist is significantly faster as compared to other approaches. However, as mentioned above, the whitelist approach is vulnerable to package renaming obfuscation technique, which is commonly used in the Android apps. Furthermore, despite their attempt to overcome

the incompleteness of whitelist by studying a large number of apps, many of the less popular libraries are still not listed. Since the study is based on a large number of apps, it may be expensive to keep the whitelist up to date. Similarly, LibRadar requires a time-consuming pre-processing step of extracting unique features from a large number of apps and clustering them, which can be expensive to update their database. This can be a problem because the Android ecosystem is fast paced and dynamic, new libraries will be emerging at a fast pace. For `LibSift`, the total time taken to TPLs in all 300 apps is slightly shorter than LibRadar (after the database is established). Currently, LibRadar is able to provide the users with more details, such as meaningful package name for those obfuscated TPLs. However, there are no obvious challenges that prevent `LibSift` from including these features in the future and it is part of our plans for future work.

4.4 Threats to Validity

As there are currently no conventions to determine whether a part of a software program is actually a TPL, the validity of our assumption that each module is a TPL could be threatened. This can happen when a library contains separated modules that are independent of each other. However, we believe that the negative consequences of missing a library are much greater than splitting up a library. Note that this is also an issue for both the state-of-the-art approaches. Furthermore, it is possible that our reported module contains more than one library that have high dependencies on each other. However, TPLs are designed to work individually and this situation should rarely occur. In addition, our approach checks for package homogeneity before merging the packages into module, thus reducing the possibility this error.

Our analysis is limited to free Android apps and could threaten the validity of the generalization of our findings. For instance, it is very likely that commercial apps have more obfuscated modules to protect their interests. Furthermore, the size of our dataset is a very small amount compared to the millions of apps available across the Android markets. However, to mitigate the threat, we use real-world dataset which covers the top popular apps from different categories.

4.5 Related Work

In this section, we discuss the related work in the existing literature and we divide this section into three parts. We first discuss the related work that involves the detection of TPLs in Android apps to achieve their goals. Secondly, we discuss the work that employs module decoupling technique. Lastly, we discuss the related work that aims to identify TPLs in Android apps.

Several Android app analytic tasks require the identification and removal of TPLs in the apps as they may affect the accuracy and efficiency of the analysis. They typically do so via whitelist matching or code based analysis. For example, Chen et al. [19] use a whitelist to remove apps that use same framework or common libraries from their app clone detection. Aafer et al. [72] use a whitelist to remove any APIs exclusively invoked by TPLs and improve the accuracy of their Android malware detection. DroidMOSS [30] reduces false positives in their app clone detection by removing features from advertisement libraries using a whitelist. Instead of whitelist, Wukong [39] uses the frequency of different Android APIs calls in each sub-package as a feature and performs strict comparison to cluster identical package for identifying and filtering TPLs, before detecting Android app clones. DNADroid [51] identifies TPLs by comparing the SHA-1 hashes of known libraries and excluding them from their Android app clone detection. Andarwin [76] uses program dependency graph and clustering techniques to eliminate TPLs and detect semantically similar Android apps. Glanz [67] first identify and remove library code in apps based on the analysis of abstracted code representation. They then perform fuzzy hash on the remaining app code to detect repackaged apps. Chen et al. [75] first cluster similar packages from a large dataset of apps to identify libraries, then analyze them to find potentially harmful libraries. We believe that the results of these studies can be further improved by using our approach to identify TPLs.

The following studies employ module decoupling technique on Android apps to address different challenges and prove that module decoupling technique works well on Android apps. PiggyApp [42] aims to detect piggybacked apps (legitimate apps with malicious code attached), by first separating the app code into primary and non-primary modules and comparing apps with similar primary modules to identify repacked app with rider code. Addetect [77] decouples Android app into individual modules, they then extract features from these modules and use machine learning

technique to identify advertisement libraries. Droidlegacy [78] partitions Android apps into loosely coupled modules and comparing the signature of each module to known malware families to identify piggybacked malicious apps. These studies perform module decoupling at different granularity levels. In the module decoupling performed by PiggyApp, each node in the graph represents a Java package that includes all the Java class files declared within it. However, AdDetect only considers the packages that represent the root of the package subtrees. Whereas in DroidLegacy, each node in the graph presents a Java class. In `LibSift` we perform module decoupling at the same granularity level as PiggyApp.

The following studies focus on detecting TPLs in Android apps. Li et al. [79] first build instances of potential libraries based on the apps' organizations and primary relations information. They then generate obfuscation resilient features from each instance and cluster instances with equivalent features as TPLs based on a predefined threshold of occurrence frequency. Backes et al. [37] generates TPL profiles based on class hierarchy information from a comprehensive library database and the profiles are used to match exact copy of the same library version. Li et al. [3] identify TPLs in Android apps from a large dataset of 1.5 million apps. Their approach is based on the appearance frequency of the package name in the large dataset and subsequent refinements to refine the list. LibRadar [2] extends Wukong's [39] clustering-based technique by performing feature hashing on the features to effectively detect TPLs used in Android apps. The techniques that are based on the assumption that TPLs are used in a large number of apps may not be able to identify less popular or new TPLs.

4.6 Conclusion and Future Work

In this chapter, we have presented our approach to detect third-party libraries in Android apps. Our approach is based on the observation that libraries in software program are highly cohesive but loosely coupled. Therefore, for a given Android app, we perform module decoupling based on its P_kDG and identify its primary and non-primary modules. We have implemented a prototype of `LibSift` and evaluated it on a set of real-world Android apps. Our result shows that `LibSift` is able to effectively identify TPLs in Android apps. We have also compared our approach with two other state-of-the-art approaches, LibRadar [2] and whitelist

from Li et al. [3]. The results show that our approach can detect libraries not detected by them.

Despite the good results, `LibSift` can still be improved in multiple ways. For our future work, we plan to extend our current work by improving on the current algorithm and providing additional useful features. For example, it may be useful to recognize and identify the obfuscated TPLs. Furthermore, additional information such as the type of library and the maliciousness of the library are also important for certain program analysis tasks. All these can be achieved by performing static analysis on the code of each of the modules identified by `LibSift`.

Chapter 5

Machine Learning Based Android Malware Detection in Face of Concept Drift: Empirical Study and Recommendations

5.1 Introduction

Recent research has established that with the raise in popularity of Android apps, Android malware are also growing rapidly and at the same time evolving with more sophisticated attacks [11–14] and evasion techniques [15, 16]. Symantec [80] revealed that variants per Android malware family has increased by more than a quarter in year 2016.

Machine Learning based malware detection. Over the last decade, Android malware detection and analysis has evolved as one of the most challenging problems in cybersecurity. Researchers from both academia and industry have invested significant effort into designing accurate and scalable approaches to address the same [50, 71, 81–83]. One of the most prominent classes of Android malware detection approach is the machine learning (ML) based approach. Typically such an approach could be perceived as a three step process: (1) a sufficiently large dataset of malicious and benign apps is collected, (2) semantic/syntactic features

that characterize malice behaviors are extracted from these apps through static/dynamic analysis, and (3) an off-the-shelf ML classifier (e.g., Support Vector Machine (SVM), Random Forest (RF), etc.) is then trained/evaluated on this dataset. These approaches have focused on distinguishing between malware and benign apps. In other words, they model malware detection as a binary classification problem, whereby there are only two class labels, malware or not malware (i.e., benign). Despite the importance of such work which contributes as a step towards addressing the complications associated with malware, it is not enough to simply detect and remove the identified malware.

After malware has been detected, having information on the malware type or the family to which the malware sample belongs is beneficial to the malware forensic analysts. The reasons are two folds: (1) having knowledge on the malware family can help the analyst in assessing and mitigating the damage done by the malware (e.g, identifying residual malicious components or determining if any data has been leaked). (2) Albeit manual analysis being obligatory in providing in depth information for the malware behavior, it is impractical to manually analyze all available malware samples. Given the knowledge on the malware families, only a few samples in each semantically similar malware cluster need to be analyzed. To this end, existing studies [11, 20, 78, 84, 85] have proposed to automatically classify malware into families based on ML techniques. In this scenario, the problem is typically model as multiclass classification problem, where the class labels correspond to the malware families.

Batch learning models. The aforementioned process through which the detection models (including the malware family classification models) are built is often referred to as *batch learning* i.e., the model is trained with a batch of labeled malicious and benign samples, or in the case of malware family classification it is trained with a batch of malicious samples labeled with their respective family. An overwhelming majority of the existing Android malware detection approaches follow this learning process. To throw light on such representative studies, we refer the reader to the following: static analysis based approaches: [43, 71, 82, 86], dynamic analysis based approaches: [81, 87], hybrid analysis based approaches: [83].

Limitations of batch learning. Recent studies such as [88] categorically establish that the two challenges described below make batch learning particularly unsuitable for real-world malware detection.

(1) Malware evolution & concept drift. In batch learning, an important underlying assumption is that the probability distribution of the extracted features in the data source are stationary (i.e., does not change over time). However, on the contrary, this is not the case for malware as they constantly evolve due to several reasons, such as to evade detection, exploiting newly found vulnerabilities or newly introduced Android features, etc. As a result of this evolution, new malware features emerge, gain prominence and fade off, over time. The change in the distribution of the malware features over time caused by evolution, is known as 'concept drift" in the ML literature [89, 90]. Formally, it is defined as any change in the conditional probability distribution $P(C|X)$ over time. Concept drift makes the collection of malware identified today unrepresentative of the ones generated in the future. Hence, in the face of concept drift, the batch learning models trained on a dataset at a particular point in time are rendered gradually obsolete over time.

(2) Scalability. One way to prevent the batch learning models from becoming obsolete is to retrain them periodically with fresh up-to-date datasets. However, as mentioned earlier, the volume of malware data grows at an alarming rate. Hence, retraining the models with such huge volume of samples would pose severe scalability issues.

Having explained the peculiar concerns of malware evolution and its consequent challenges, in this work, we intend to systematically evaluate, in the face of concept drift, whether the state-of-the-art Android malware detection approaches perform as well as they claimed. When they fail to do so, we propose modifications which enable them to function both accurately and efficiently in the real-world malware detection setting with significant concept drift. More specifically, we reimplemented and make publicly available two static analysis based approaches which produce excellent results in terms of both accuracy and scalability in the batch learning setting, namely, `Drebin` [43] and Allix et al. [86]. We observed that in face of concept drift, the limitations of these batch learning models are that they are unable to handle streaming features, samples and classes that reflects the real world settings. To this end, we suggest and evaluate on three technique agnostic modifications namely, feature hashing, online learning and progressive learning to address the limitations. By performing a series of experiments with both of the approaches on large real-world dataset, we hope to gain insights for successful designing of ML based malware detection approach. The dataset consist of more

than 80,000 apps, inclusive of both dated and modern samples that spans from year 2009 to 2016.

Contributions. We make the following contributions in this chapter:

- We use two state-of-the-art existing malware detection approaches with orthogonal choice in feature sets to demonstrate the limitations of existing batch learning ML based malware detection and malware familial classification approaches in the face of concept drift.
- We suggest technique agnostic modifications to existing batch learning ML based malware detection and malware familial classification approaches and demonstrate using two such state-of-the-art that the suggested modifications significantly improve their effectiveness and efficiency.
- We reimplemented `Drebin` [43] and Allix's approach [86] and make them publicly available.

5.2 Preliminaries

ML techniques has been widely adopted to address security threats in software due to their capability to learn accurate detection model automatically and thus avoiding the need for the laborious task of crafting detection patterns manually. ML is a promising technique that has been proven to be capable of providing solutions to multiple challenges. Thanks to the various ML tools and libraries that are easily accessible online, it is quite straight-forward to apply ML techniques on most problems. Despite so, the ML algorithms are difficult to fully understand and as a result, they are often used as black-box. This leads to the situation where the results obtained from ML based techniques are often accepted without much scrutinizing. However, recent studies [86, 91] have begin to question whether these experiments that are conducted in a *"in the lab"* scenario are valid in the real-world scenario.

To further understand the state of issue, in the following subsections, we first discuss on the working mechanism of ML based Android malware detection approaches in general. Then in greater details, we introduce two state-of-the-art approaches

that we will be using to demonstrate the limitation of current approaches and evaluate our recommended solutions. Lastly, we discuss on app evolution and the challenges that they may impose on ML based malware detection approaches.

5.2.1 ML Based Malware Detection Approaches

The success of ML based approaches is dependent on a number of key factors. Indeed, one of the well known working principle of ML algorithm is "garbage in, garbage out", nevertheless, given high quality input, superior results can be achieved. In particular, to achieve high malware detection accuracy, the ML algorithm should be fed with high quality ground truth data and relevant features that capture the characteristics of malware. More details on both the key factors affecting the performance of the ML approach are elaborated below.

Reliable ground truth. In the case of malware detection the ML algorithms learn what is malware based on the given labeled samples. Therefore, the availability of reliable ground truth dataset for both benign and malware is important for training the classifier. Existing research [86] has demonstrated that the models which had achieved more than 90% in terms of F-measure when trained on are reliable ground truth, had their F-measure diminished to nearly 0% when trained on unreliable benign apps. However, it is challenging to assemble a sufficient number of reliable ground truth data as large amount of manual effort is required to verify the claim. As a result, existing literature typically either use malware collections that are shared by other research groups, or establish the ground truth for their apps based on the votes of multiple anti-virus scanners hosted on VirusTotal [65]. This may lead to the situation where malware samples in the evaluation dataset is significantly older than the benign samples. When this happens, the results obtained may represent the ability of the approach to distinguish between samples from different age instead of between malicious and benign [91]. Thus, when the model is put to practice in the real-world, it performance will degrade.

Furthermore, the ML based malware detection approaches in literature often follow the batch learning model which assumes that the underlying probability distribution of the extracted features does not change over time. As such, they typically train and evaluate the model on a random train and test dataset split without considering the temporal order of the training and testing samples. Nevertheless, given

the vibrant ecosystem of Android, thousands of new apps are being uploaded daily, either to the official market Google Play or to the numerous third-party markets. Hence, in reality the malware detection are often required to be deployed at the front line to perform prediction on the apps as soon as they surfaced, before they reach a large number of users, so as to minimize the damage. This implies that in reality the detection models will first be trained on historically anterior apps and the prediction will be performed on historically posterior apps. Consequently, malware detection without considering temporal order yields biased results that are artificially improved and may not be a true indicator of their performance in reality [92].

Representative feature set. Given the labeled samples, the ML algorithms typically learn to distinguish between different classes based on the observed characteristics (features) of the samples in the respective class. Several feature sets have been proposed in the existing malware detection work and they reportedly achieve considerable results [43, 71, 86].

More importantly, features that are not present in the training set cannot be understood by the typical batch learning ML algorithm. For example, to perform Android malware detection, network address may be extracted from the *apk*s and use as one of the features. However, as aforementioned, in the real-world scenario the batch learning model are trained on samples that are historically anterior and perform prediction on the historically posterior apps that stream in. The apps that stream in may contain some previously unseen features (e.g., new network address) and not being able to handle as such new information cause the detection performance of model to degrade over time.

5.2.2 State-of-the-art Approaches

To demonstrate the limitations of batch learning malware detection approaches in face of concept drift and subsequently show the advantages of our proposed modifications, two state-of-the-art approaches namely, `Drebin` [43] and Allix's approach [86] are selected from the existing literature. The reasons for choosing these two approaches are three folds. Firstly, they are both state-of-the-art approaches that have reported remarkable results from their experimentation. Secondly, it is due to

simplicity. The techniques used in both of these approaches are straightforward feature extraction and applying of ML algorithm. Last but not least, the architecture of both the approaches provide the ease of demonstrating our proposed modifications. At this juncture, we would like to clarify that the goal of this study is not to critique against their work, but to show that malware evolution can significantly impair even state-of-the-art approaches such as these two. For completeness, the key characteristics of both approaches are highlighted below.

Drebin. `Drebin` is peculiar in the sense that instead of handpicking features that are perceived to work well, it went for broad static analysis to gather as many features from an app's code and manifest as possible. The static features extracted by `Drebin` can classified into 8 semantic groups which include, hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls and network addresses.

However, these sets of features as it is, cannot be directly understood by the chosen ML algorithm. As such, before the features are input to the ML algorithm, they should first be transformed such that each app is represented in the form of a feature vector. The transformation technique used in `Drebin` is to map the features into a vector space, where each dimension is either 0 or 1. When an app contains the feature, the respective dimension is set to 1 and 0 otherwise. The joint set of all 8 feature sets reportedly contains about 545K unique features based on their dataset of over 129K apps.

More specifically, `Drebin` was experimented on the a dataset which contains 123,453 benign and 5,560 malware samples, where the malware samples are verified by at least two anti virus scanner on VirusTotal [65] and due to their ambiguity adware are removed from the dataset. The ML algorithm employed in `Drebin` is linear SVM, it is chosen by the authors of `Drebin` due to its efficiency and explainability. `Drebin` is evaluated via hold-out validation with random 66% and 33% training and testing split, the results averaged over 10 runs. Their evaluation strategy ensures that the prediction results are only based on unknown malware samples (i.e, not in training set). However, it does not address the concern of historical coherence in the training and testing samples.

Allix's approach (CSBD). The feature set used in Allix's approach [86] is string representations of all the basic blocks found in the control flow graph (CFG) of the

app which is built by performing static analysis on the app's bytecode. Each string representation is basically an abstraction of the app's code that retains structural information of the code, but discards low-level details such as variable names or register numbers. For convenience and readability we will refer to Allix's approach as CFG-Signature Based Detection (`CSBD`) in the rest of the article.

Similar to `Drebin`, the features are mapped into a vector space, where each dimension is either 0 or 1. When an app contains the basic block, the respective dimension is set to 1 and 0 otherwise. They reportedly extracted over 2.5M unique features (i.e., basic blocks) based on their dataset of over 50K apps. Due to the exceedingly large number of features, feature selection is performed to improve the computation efficiency of the downstream task and only top N features with highest Information Gain (IG) is retained. It is demonstrated that the median value of F-measure stabilizes when the number of features is above 1K and the best accuracy obtained is when $N = 5$K which is the maximum N they have evaluated.

To evaluate their approach against existing approaches, the authors experimented with four different ML algorithm namely, RandomForest, J48, JRip and LibSVM, with RandomForest achieving the best accuracy. The models are tested with the typical 10-Fold cross validation technique. Moreover, they also conducted in the wild experiment without 10-Fold cross validation but instead uses a training and testing dataset ratio that is closer to the real-world setting. However, the temporal property of the samples are not considered in both cases.

5.2.3 App Evolution

From the above, we can see that the results obtained by batch ML based approaches, such as `Drebin` and `CSBD`, are only valid with the assumption that the probability distribution of the extracted features remains stationary. However, in reality, this assumption does not hold, as apps evolve over time due to several reasons [16]. Firstly, apps may evolve due to natural influences such as adding new features, improving performance and bug fixes. In the case of malware, it would be enhancing malicious capabilities or new form of attacks (zero-day malware). Secondly, the evolution may happen due to the apps adapting to environmental changes such as changes in Android framework (i.e., new version of Android release) and changes in the libraries used by the app. Lastly, evolution due to

obfuscation (aka polymorphic evolution [93]) may occur in benign apps to protect the developers' intellectual property and occur in malicious app as a mean to evade detection.

These evolutions are all applicable to both malicious and benign app, with natural and polymorphic evolution being likely more prominent in malware due to their attempts to evade detection. Reinforcing the facts, recent studies [11–14] have noted that malware have evolved from performing simple phone cloning, sending of premium-rated SMS to more complex and disruptive malware such as ransomware, botnets and cryptolocker. In addition, [16] has reported that there is an increase in the usage of dynamically loaded code, Java reflection and native code for Android malware. As a results, a significant number of threats in more recent malware samples are not captured in the dated malware samples [4]. In the following experiments, we demonstrate that even state-of-the-art approaches degrade significantly in the face of such phenomenon.

5.3 Empirical Study Design

The goal of this study is to investigate the implications of concept drift on ML based malware detection approaches and the possible solutions to address them. Our proposed modifications are technique agnostic, however, for demonstration purposes we perform systematic evaluation on two chosen state-of-the-art approaches `Drebin` and `CSBD`. We design this study as a large scale empirical study which contains a series of experiments whereby each of them is targeted at addressing the research questions that we will introduce in the following. The results of this study provide insights to help build more successful ML malware detection approach. To this end, we reimplemented `Drebin` and `CSBD` in approximately 1,400 and 900 lines of python code, respectively and make them publicly available. Moreover, we show through experiment below, that both our implementations are in line with the original work, in terms of both accuracy and efficiency.

5.3.1 Baseline

To ensure that the inferences drawn from the following experiments are indeed due to the influence of concept drift, we should have a baseline for comparison. That is to say, we need to demonstrate that the approach can work well when concept drift is not present (i.e., batch training and testing without considering historical coherency). Then subsequently, using the same approach, examine how the performance of the approach degrades in presence of concept drift.

In light of the above, we reimplemented two state-of-the-art approaches, namely, `Drebin` and `CSBD` and show that they can achieve good accuracy in absence of concept drift. Reimplementing the approaches allows us to fully understand the implementation and provides the ease of incorporating our proposed modifications to justify the viability of our suggestions. Furthermore, by achieving similar good results as report in their original work, in turn prove that our reimplementation of `Drebin` and `CSBD` is faithful to the original work.

5.3.2 Challenges and Recommendations

(C1) Streaming features. As aforementioned, the underlying characteristic distribution captured by some features may change over time due to evolution of the apps. For example, before the commencing of actual training, `Drebin` and `CSBD` first map the extracted feature to a fixed N-dimensional binary vector space, where N equals to the number of unique features observed at the time of mapping. When a sample with concept drift (i.e., new feature) arrives, the models are unable to administer the unseen features that are not part of the unique features they are trained on. In order to address this challenge, one can set a sufficiently large vector space dimension to cater for the unknown number of unseen features. However, in practice, it is difficult to identify a good dimension for the catering, since the rate at which the number of new features grows vary from feature set to feature set. Another intuitive solution is to append the unseen features to the vector space and retrain the model. However, given the large volume of Android apps, frequent retraining of the model can be very costly.

(R1) Feature hashing. Feature hashing aka hashing trick [94] is a popular technique to address such problem. To this end, we intend to study the effect

of feature hashing on batch ML based approaches, by replacing the vectorization technique of such approaches with feature hashing technique, whereby a hashing function is applied to the features and these hash values are used as indices in the sparse matrix of a predefined dimension δ. Due to the large volume and the streaming nature of Android apps, feature hashing provides several advantages which make it well suited for malware detection problem. Firstly, it provides memory efficiency, as it allows transforming of features to a vector space of a predefined dimension, without the need to store the one-to-one feature to indices mapping for the whole dataset which may not fit into memory. This in turn enables the model to handle streaming data. Lastly, by limiting the dimension of the feature vector, it can help to improve the efficiency of the model in learning or prediction.

(C2) Streaming samples. We have mentioned in the above that in the real world use case, the training samples of a malware detection model should be historically anterior to the apps that are to be predicted (test samples). Despite so, the training and testing regiment of batch learning approaches in existing work have training and testing apps that appear in a similar time frame, or even having some training apps that are historically posterior to the testing apps [92]. Consequently, the results obtained from such experiments may not be representative of the real-world scenario.

(R2.1) Retraining. Over time, as the drift become more significant, the typical batch learning models that are trained with only historically anterior apps will have greater difficulty in detecting the historically posterior malware samples that have evolved over time. To overcome the challenge, existing studies [95, 96] have suggested to retrain their models periodically. There are two general approaches to retrain the model [97]. The first approach is to simply retrain the model at fixed intervals. This approach does not require any technical analysis (i.e., concept drift tracking) to be performed beforehand. However, in the case of Android malware detection, retraining the model frequently is a costly operation due to the large volume of Android apps, and it may not be worth the effort as the enrichment to the classifier may not be significant. On the other hand, infrequent retraining will increase the period of time where the model is less trustworthy. Therefore, the second approach tries to identify changes in the population distribution by monitoring indicators such as parameters or performance of the classifier, and

properties of the features [98]. The model will then be updated when significant changes are observed in the indicators.

(R2.2) Online learning. Recent work [99] have demonstrated that online training can significantly improve malware detection performance in terms of both scalability and efficiency. Typically, an online training model continuously adapt to each labeled sample it received and make prediction of a new sample based on the updated model. This provides several advantages over the typical batch learning model. The batch learning models generally optimize their parameters by taking multiple passes over the training samples which also lead to high memory consumption during the training. However, unlike the batch learning models the online learning models take exactly one pass over the training samples, thus leading to high efficiency and low memory requirement. Furthermore, the learning mechanisms of online training models allow for natural adaption of the drift in malware, which makes it well-suited for malware detection.

In light of the above, we intend to study the extent to which online learning model can enhance the capabilities of batch ML based approaches in adapting to concept drift and maintain high detection accuracy over time. To this end, we replace their detection model with online learning model.

(C3) Streaming classes. As mentioned in Section 5.1, simply detecting malicious apps is not sufficient, upon successfully detecting a malware, it is important to identify the malware type/family of the detected malware sample for further analysis. In order to cope with the rate at which malware variants are created, it is clear that the process to categorize the malware into groups which correspond to their families needs to be automated. Similar to ML based malware detection approaches, existing approaches which attempt to classify malware samples into their families typically do not consider historical coherency. Moreover, they require a priori information on the number of classes and assume that it will remain as such. Thus, limiting their ability to handle real-time streaming samples, where over time the model may encounter entirely new malware families that are not observed in the training dataset (i.e, a new class).

(R3.1) Retraining. Similar to the binary classification problem above, the challenges of streaming classes may be addressed by retraining the model. This is

because retraining of the model allows it to be updated with knowledge on samples belonging to the new classes in the updated training set used to retrain the model.

(R3.2) Progressive learning. Note that despite being able to effectively handle streaming samples, the online learning algorithms, such as the Passive Aggressive algorithm, are unable to handle streaming classes. A learning paradigm known as progressive learning [100] is proposed in the literature to address this challenge. Progressive learning is basically an upgraded online learning method that is not only able to handle streaming samples (concept drift), but also streaming classes. As and when a sample belonging to a yet unseen class arrive at the model, the neural network model increase in size and its interconnections and weights are redesigned to adapt to the new information. The additional classes are learnt in conjunction with the existing knowledge like they existed at the beginning.

We propose to extend the capabilities of `Drebin` and `CSBD` in adapting to streaming classes by replacing their batch learner classifier with progressive learner classifier.

5.3.3 Research Questions

In light of the above, we introduce the research questions that we have formulated to study the challenges and viability of our proposed modification for ML based malware detection approaches in face of concept drift.

RQ1. Reproducibility

> **RQ1.1.** *How accurate are* `Drebin` *and* `CSBD` *when performing malware detection on dataset with timeline similar to the ones used in the original work and not considering historical coherence?*

> **RQ1.2.** *How efficient are* `Drebin` *and* `CSBD` *when performing malware detection on dataset with timeline similar to the ones used in the original work and not considering historical coherence?*

RQ2. Streaming Features

> **RQ2.1.** *How does feature hashing affects the accuracy of* `Drebin` *and* `CSBD`*?*

RQ2.2. *How does feature hashing affects the efficiency of* `Drebin` *and* `CSBD`*?*

RQ3. Streaming Samples

RQ3.1. *How does considering history coherency affects the malware detection performance of* `Drebin` *and* `CSBD`*?*

RQ3.2. *How does retraining at different intervals influence the performance of* `Drebin` *and* `CSBD`*?*

RQ3.3. *How does replacing their batch learning model with online learning model influence the performance of* `Drebin` *and* `CSBD`*?*

RQ4. Streaming Classes

RQ4.1. *How does considering history coherency affects the performance of* `Drebin` *and* `CSBD` *on malware familial classification?*

RQ4.2 *How does retraining influence the performance of* `Drebin` *and* `CSBD` *on malware familial classification?*

RQ4.3 *How does replacing their batch learning model with progressive learning model influence the performance of* `Drebin` *and* `CSBD`*?*

5.3.4 Dataset

To facilitate the study, it is necessary to have a set of malicious and benign apps from various timeline. Therefore, we collected from multiple sources, over 80K real-world benign and malicious Android apps that span over a timeline of seven years. A summary of the dataset used in our experiment is presented in Table 5.1. More specifically, 51,991 benign apps are collected from various Android app markets (i.e.,Google Play [101], Anzhi [102], AppChina [103], SlideMe [104], HiApk [105], FDroid [106] and Angeeks [107]). We verify the apps using VirusTotal [65], an online malware detection service hosting various (up to 60) anti-virus scanners, the

TABLE 5.1: Dataset from different timeline

Category	Source	# of samples	Time of compilation
Malware	Drebin	5,560	2009 - 2012
	AMD	24,650	2009 - 2016
Benign	Google Play	46,800	2009 - 2016
	Anzhi	2,957	
	AppChina	1,845	
	SlideMe	289	
	HiApk	65	
	FDroid	29	
	Angeeks	6	

benign set only contains apps that are not flagged by any of the anti-viruses as malicious. Their date of creation span from year 2009 to year 2016.

Furthermore, malware datasets from two different sources are used in our experiments. The first malware dataset is `Drebin` dataset, that consist of the 5,560 `Drebin` malware, including malware samples from 179 malware families, that spans from year 2009 to 2012. The second malware dataset is AMD dataset, a malware collection from [4] which consist of 24,650 apps, spanning from year 2010 to 2016. It includes malware samples from a total of 71 malware families. In addition, a malware can achieve its malicious objective through different means and behavior. Based on this, malware samples can be categorized into semantically different groups known as varieties. The AMD malware collection can be divided into 135 varieties.

Since both the chosen state-of-the-art approaches, `Drebin` and `CSBD`, evaluated their approach on malware genome [50] dataset which is dated from August 2010 to October 2011, for simplicity, we consider all malware samples before 2012 as dated malware and malware samples from year 2012 and beyond as modern malware.

5.3.5 Feature Construction

The raw features extracted from the apps cannot be directly understood by the ML algorithm and thus required to be transform to feature vectors. Therefore, we map the extracted feature sets to a N-dimensional binary vector space, according to the original work of `Drebin` and `CSBD`. For all apps in our dataset, the total number of unique features extracted for `Drebin` and `CSBD` is approximately 564K and 10.43M

respectively. Due to the exceedingly large number of features extracted by `CSBD`, the authors noted the need to reduce the number of features to improve computation efficiency for the downstream tasks. They performed feature evaluation based on the IG measure and retain only the best N features. Moreover, they conducted a study on how would the number of features (in the range of 50, 250, 500, 1K, 1.5K, 5K) impact the accuracies of the models and concluded that the F-measure improve with the number of features and since the best detection accuracy is reported when the number of features is 5K, we use the same in all our experiments for `CSBD`.

5.3.6 Evaluation Metric

For binary classification experiments, standard measures such as precision, recall, F-measure and Cumulative Error Rate (CER) are used to determine the detection accuracy of the model. These measures are in the range between 0 to 1, where higher values of precision, recall, F-measure and lower values of CER indicate better detection accuracy. Whereas, for multiclass classification, macro and micro average of these measures are used. On the other hand, for clustering experiments, Adjusted Rand Index (ARI) and Normalized Mutual Information (NMI) are used to determine the quality of clusters. ARI is in the range between -1.0 to 1.0 while NMI is between 0.0 to 1.0 and in both cases higher values indicate better clustering quality. Efficiency of the model is evaluated based on the time taken (in seconds) for training and testing.

5.3.7 Experimental Setup

All the experiments are conducted on a server with 20 cores of Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and 128 GB ram running Ubuntu 16.04.

5.4 Results and Discussions

We have introduced our research questions in Section 5.3.3, in this section, we present and discuss on the results of the sets of experiments targeted at addressing these research questions.

TABLE 5.2: Accuracy of `Drebin` vs `CSBD` on dated malware samples - averaged (±std) over 5 runs

Approach	**Precision(%)**	**Recall(%)**	**F-measure(%)**
`Drebin`	97.52 (±0.41)	98.32 (±0.31)	97.92 (±0.22)
`CSBD`	94.61 (±0.55)	96.85 (±0.30)	95.71 (±0.27)

5.4.1 RQ1. Reproducibility Analysis

In this RQ we aim to establish a basis to evaluate the implications of concept drift and subsequently the viability of our proposed modifications, such as feature hashing, online learning and progressive learning. As such, with our implementation of state-of-the-art approaches, we aim to reproduce the performance in terms of accuracy and efficiency achieved in the original work. By verifying through this experiment that we can achieve results similar to the original work, it also demonstrates that our implementation is in line with the original work.

Experiment design. Since both `Drebin` and `CSBD` have evaluated their approaches on malware samples up to year 2011, in this experiment, we perform malware detection for `Drebin` and `CSBD` on a set of 4,944 malware samples and an equal number of benign samples that are developed before year 2012. Similar to the original work, the training and testing processes are conducted in a batch fashion. 70% of the samples in this set are chosen at random and used for training the classifiers, while the remaining 30% are used for testing. The classifiers hyper-parameters are tuned based on the training set with 5-fold cross-validation, whereas the test set is only used for determining the prediction accuracy. This process is repeated 5 times and the average results are reported.

Their efficiencies are evaluated based on the training and testing duration, where training duration includes the time taken for feature transform and feature selection (only performed for `CSBD`), since they are necessary process for training the models. The number of features extracted are also reported alongside the training and testing duration, as it strongly influence the efficiency of both processes.

Results and discussions. From Table 5.2, we can observe that our implementations of both `Drebin` and `CSBD` can reproduce the high accuracy as reported in their original work. Note that direct comparison of the readings with the original work is not meaningful as different datasets are used in the experiments. However,

TABLE 5.3: Efficiency of `Drebin` vs `CSBD` on dated samples

Approach	# of feat.	# of feat. after selection	Training duration (seconds)	Testing duration (seconds)
`Drebin`	∼50K	NA	1.84	0.1767
`CSBD`	∼1.37M	5,000	612.58	10.26

based on these results, we can conclude that the results of `Drebin` and `CSBD` are reproducible albeit under different settings.

Although not the focus of this paper, it is interesting to note that from Table 5.2 that the number of features extracted by `CSBD` is significantly larger than `Drebin`, as this difference may affect how our recommended solutions affect the approach. The difference in the number of unique features observed in this experiment compared to the original work is due to a different and lesser amount of apps used in this experiment. Furthermore, `Drebin` clearly outperforms `CSBD` in terms of training efficiency. In addition to the large difference in the number of features, the fact that `Drebin`'s linear model (Linear SVM) can be trained in a shorter amount of time than quasi-linear models (Random Forest), would also have contributed to the difference in the training and testing time.

5.4.2 RQ2. Streaming Features

In the real-world malware detection scenario, the malware detection models are first trained on samples that are historically anterior to the test samples that stream in. Furthermore, the streaming samples may contain an unknown amount of previously unseen features due to malware evolution. Feature hashing is a well known technique typically used to handle such situation. In this RQ, we investigate the advantages and disadvantages of applying feature hashing to batch learning approaches for Android malware detection.

Experiment design. In order to illustrate the impact of feature hashing to a greater extend, we use the whole dataset of over 80K apps in this experiment. Similar to RQ1, the experiments are performed with 70% training and 30% testing split, and 5-fold cross-validation is used to tune the hyper-parameters. With the results averaged over 5 runs.

TABLE 5.4: Accuracy of `Drebin` vs `CSBD` with and without feature hashing

Approach	Feature hashing	Precision(%)	Recall(%)	F-measure(%)
`Drebin`	No	97.53	98.24	**97.88**
	Yes	**97.83**	97.80	97.81
`CSBD`	No	96.80	98.07	97.43
	Yes	96.20	**98.45**	97.32

TABLE 5.5: Efficiency of `Drebin` vs `CSBD` with and without feature hashing

Approach	Feature hashing	# of features	# of feat. after selection	Training duration (seconds)	Testing duration (seconds)
`Drebin`	No	425,269	NA	22.25	1.23
	Yes	50,000	NA	18.36	0.93
`CSBD`	No	8,164,447	5,000	9,211.91	196.13
	Yes	50,000	NA	12,688.82	171.18

To investigate the effects of feature hashing, we conduct two sets of experiments. We first perform the experiment with the default `Drebin` and `CSBD` models to establish the basis for evaluating the models with feature hashing. On top of that, we perform another experiment with the same setup, but this time round the binary vectorization for the default models are replaced with feature hashing. We arbitrarily chose $\delta = 50$K. Note that for `CSBD` with feature hashing, feature selection is not performed.

Results and discussions. From Table 5.4, it is clear that the effect of feature hashing on the accuracies of both the approaches are negligible. More specifically, when `Drebin` with approximately 425k features is compressed to just 50K features through feature hashing, the difference in F-measure is 0.07%. In the case of `CSBD`, 5K top features selected from 8.164M features compare to compressing to 50K features from the same through feature hashing, the difference in F-measure is 0.11%. This shows that despite the possibility of hash collisions, where two different features are assigned to the same index, it have little impact on the accuracy of the models. Reinforcing this fact, despite the significantly higher risk of collision, the decrease in F-measure is just 0.04%. This is likely because the feature domain is very sparse and relatively few of them tend to be informative or are rare words, overall any hash collisions will mostly impact less informative features and not affect the decision of the models considerably.

From Table 5.5, we can observe that feature hashing offers slight improvement in terms of time efficiencies. For `Drebin` the training and testing time is reduced by 17% and 24%, respectively. On the other hand, the training time for `CSBD` increased by 0.38% while testing time is reduced by 13%. A deeper look into the training time of the `CSBD` models, reveals that the the increase in training time for the `CSBD` with feature hashing is mainly contributed by the RF algorithm, due to larger number of features (i.e., 5K top features to 50K features).

5.4.3 RQ3. Streaming Samples

In this experiment, we attempt to investigate the effects of concept drift on different batch ML based malware detection techniques, when the apps historically posterior to the training apps stream in, which mimics the real-world setting. In addition, we further investigate the adaptiveness of both the models, in terms of accuracy and efficiency, to some of the possible solutions to address concept drift.

Experiment design. In light of the above, for each of the two state-of-the-art appraoches we train one vanilla and three variants which sum up to eight models in total. These models are evaluated on streaming samples that are historically posterior to the training samples. These streaming samples are temporally sorted according to their timeline starting from year 2012 and ending with 2016.

To study the effect of concept drift on the vanilla models of `Drebin` and `CSBD`, they are only trained once on samples with timeline before year 2012, and tested on all streaming samples without retraining.

The next two pairs of variants aim to study the adaptiveness of approaches to the periodical model retraining solution. These two pairs of variants differ in the interval between retraining of the models, such as annually and semi-annually. Similar to the above, these variants are first trained on dated samples (before year 2012) then start to predict the samples that streams in. However, after the models have predicted all samples within the respective interval, the predicted samples are added to the training set and used to retrain the models, before moving on to predict the next set of streaming samples in the next interval. The process is repeated until all samples (up to first half of year 2016) have been predicted.

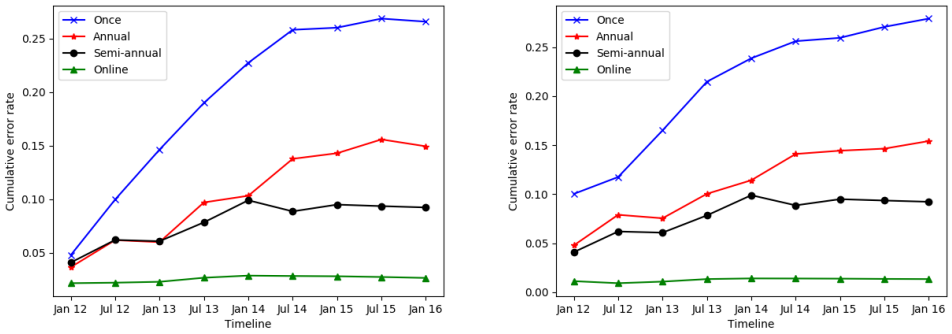
TABLE 5.6: `Drebin` streaming samples efficiency

Year (FH/SH)	# of samples	Drebin_Once			Drebin_Annual			Drebin_Semi_Annual			Drebin_Online		
		# of feat.	Train time	Test time	# of feat.	Train time	Test time	# of feat.	Train time	Test time	# of feat.	Train time	Test time
2012(FH)	9,888	67,760	1.20	1.17	77,950	0.97	1.24	78,027	0.91	1.22	50,000	0.0212	1.55
2012(SH)	16,114	67,760	-	1.43	77,950	-	1.55	120,881	1.44	1.50	50,000	0.0276	1.91
2013(FH)	23,750	67,760	-	0.93	170,529	2.65	1.00	172,240	2.17	0.98	50,000	0.0198	1.21
2013(SH)	28,552	67,760	-	1.77	170,529	-	1.89	202,002	2.76	1.84	50,000	0.0501	2.32
2014(FH)	37,726	67,760	-	1.44	266,828	4.70	1.55	263,782	3.89	1.52	50,000	0.0433	1.89
2014(SH)	45,190	67,760	-	1.33	266,828	-	1.42	316,352	5.08	1.37	50,000	0.0289	1.73
2015(FH)	52,078	67,760	-	0.22	357,832	7.44	0.72	355,100	6.30	0.24	50,000	0.0036	0.29
2015(SH)	53,248	67,760	-	0.47	357,832	-	0.49	360,109	6.36	0.49	50,000	0.0058	0.60
2016(FH)	55,656	67,760	-	0.41	371,694	8.31	0.44	375,660	6.67	0.43	50,000	0.0026	0.53

TABLE 5.7: `CSBD` streaming samples efficiency

Year (FH/SH)	# of samples	CSBD_Once			CSBD_Annual			CSBD_Semi_Annual			CSBD_Online		
		# of feat.	Train time	Test time	# of feat.	Train time	Test time	# of feat.	Train time	Test time	# of feat.	Train time	Test time
2012(FH)	9,888	1.7M	94.71	1,915.05	1.9M	143.00	1,893.47	2.0M	85.73	1,823.61	50,000	0.0173	103.33
2012(SH)	16,114	1.7M	-	2,199.26	1.9M	-	2,314.09	2.9M	133.81	3,876.09	50,000	0.0217	164.28
2013(FH)	23,750	1.7M	-	1,389.79	4.0M	216.85	2,201.34	3.9M	438.05	2,579.78	50,000	0.0243	111.38
2013(SH)	28,552	1.7M	-	2,640.11	4.0M	-	4,262.28	4.4M	277.72	5,449.65	50,000	0.0583	198.50
2014(FH)	37,726	1.7M	-	2,368.71	5.4M	396.56	5,862.88	5.5M	467.53	5,027.30	50,000	0.0420	171.68
2014(SH)	45,190	1.7M	-	2,105.73	5.4M	-	5,679.18	6.2M	363.85	6,211.98	50,000	0.0272	116.26
2015(FH)	52,078	1.7M	-	364.21	6.8M	469.74	841.17	6.7M	464.62	823.01	50,000	0.0030	14.99
2015(SH)	53,248	1.7M	-	750.23	6.8M	-	1,711.23	6.8M	444.31	1,654.20	50,000	0.0053	31.85
2016(FH)	55,656	1.7M	-	689.79	7.0M	499.43	1,666.96	7.1M	453.79	2,103.68	50,000	0.0048	31.63

Lastly, to study the adaptiveness to online learning model, we replace `Drebin` and `CSBD` batch learner classification model with an online Passive-Aggressive (PA) classifier. Recall that the original vectorization technique is not viable in this scenario, hence we use feature hashing for both the online models. Similar to the above, we first train the online models on dated samples, then keep testing and updating the model from the samples that stream-in thereafter.



(A) `Drebin` batch learning vs online learning CER

(B) `CSBD` batch learning vs online learning CER

FIGURE 5.1: Cumulative error rates for (a) `Drebin` and and (b) `CSBD` on modern malware without retraining, with annual and semi-annual retraining and online training

Results and discussions. Figure Figure 5.1 shows the cumulative error rates (CER) of `Drebin` and `CSBD` with different training models (i.e., without retraining, with annual and semi-annual retraining and online training). Furthermore, the number of features, training duration and testing duration of both the approaches are as presented in Table 5.6 and Table 5.7, respectively. The followings inferences are drawn from the results.

From Figure 5.1, it is clear that for both approaches when their models are only trained once with samples from before year 2012, their detection accuracies degrade rapidly over time. This shows that randomly splitting the training and testing set without considering historical coherence, which allows malware samples in the training set to be historically posterior to the samples in the test set, leads to biased results. The results are unsurprising since the models are unable to handle the rapidly increasing unseen unique features that they encountered when evolved samples streams in. For instance, in Table 5.6 and Table 5.7 we can see that the number of unique features the models will encounter in year 2016 compared to year 2012 are more than 4 folds larger. We believe that, given the exceedingly large volume of apps in the real-world, the increase in the number of uniques features will be more excessive.

Similarly, we also observed that for both `Drebin` and `CSBD` the CER curves of the variants with retraining is significantly less steep as compared to the ones without retraining. Moreover, we can see that the improvement is more significant when the models are retrained more frequently. These imply that retraining of the detection model can indeed help to mitigate the effects of malware evolution and that higher retraining frequency is necessary to achieve desirable accuracies.

Furthermore, still referencing from Figure 5.1, we can clearly see that both online variants perform significantly better than their batch counterpart with and without retraining. This is mainly because their online variants can instantaneously adapt to the drift and increase their capabilities to accurately predict the following samples.

In terms of overall training efficiency, it is evident from Table 5.6 and Table 5.7 that retraining of the batch models is an incrementally costly operation, not only due to the increasing number of samples added to the training set, but also the increasing feature space dimension. Given the same number of apps streaming

in, retraining the models semi-annually in comparison to annually increase the total training duration by approximately 83% and 118%, for `Drebin` and `CSBD`, respectively. Note that, since we only have samples for the first half of the year, to provide a fair comparison between annual and semi-annual retraining we do not consider the training duration for year 2016 in the previous statement. Despite updating themselves after every sample, the online models took significantly lesser training time compare to their annual retraining counterpart. This is because, the batch learner algorithms require to be retrain on the whole dataset to learn new features, while online learning models only make a small change to the weight matrix to avoid wrong prediction in the future. In addition, since feature hashing technique is applied to the online learning variants, the vectorization of the sample features are already performed beforehand for the prediction and can be reused in the updating process. This is a big advantage for approaches such as `CSBD` which have large volume of features and require much longer time to process them.

In terms of overall testing efficiency, the effect differs from one approach to another. For instance, there is no obvious difference in testing duration among the batch learner variants of `Drebin`, while its online variant took slightly longer. On the other hand, for `CSBD`, the testing duration for its batch learner variants increases with the frequency of retraining, due to the nature of the RF algorithm, while its online variant took significantly less testing duration.

To summarize, the limitations of batch learning models in face of concept drift is evident. Solution wise, frequent retraining of the batch learning models can improve the detection accuracy moderately, but it is very resource intensive. Lastly, online learning provides highly effective alternative to retraining, and at a much lower operational cost.

5.4.4 RQ4. Streaming Classes

In this experiment, we investigate the viability of state-of-the-art approaches to perform malware familial classification in a setting which mimics the real-world scenario where malware samples historically posterior to the training samples stream in to the model for prediction. This setting give raise to the possibility of encountering samples belonging to new classes of malware families that are not observed

in the training set. We further study the adaptiveness of the approaches to some of the possible solutions to address this challenge.

Experiment design. In this experiment on malware familial classification, we use the AMD dataset from [4], as it contains malware samples from over 70 malware families which exist between different periods, spanning across a total of 6 years.

To investigate the viability of `Drebin` and `CSBD` in addressing the malware familial classification problem, we replace the binary classification model of `Drebin` and `CSBD` with their multiclass classification counterpart, namely, multiclass SVM and multiclass RF, respectively. The models are trained only once on malware samples with timestamp before year 2012. Following that, the models are tested on the remaining malware samples without retraining.

To study how retraining at different interval would affect `Drebin` and `CSBD` in performing malware familial classification, we train two more variants for each of the approaches. The models are first trained on malware samples with timestamp before year 2012 and begin predicting on the temporally ordered malware samples that streams in thereafter. When the models have predicted all samples within the respective interval, the predicted malware samples are added to the training set and used to retrain the models, before moving on to predict the next set of streaming malware samples in the next interval. The process is repeated until all malware samples in the dataset have been predicted.

Lastly, to study the adaptiveness of `Drebin` and `CSBD` to progressive learning model, we train another variant of both approaches. In this pair of variants, we replace their default classifier with a progressive learner classifier [100]. Following that, the models are trained on malware samples with timestamp before year 2012. The models then keep on predicting the samples and updating itself on the remaining temporally ordered malware samples that stream in thereafter.

Results and discussions. Figure 5.3 shows the average macro and micro F1 score of `Drebin` and `CSBD` with different variants of training models (i.e., without retraining, with annual and semi-annual retraining and progressive learning). Furthermore, the training duration and testing duration of `Drebin` and `CSBD` are as presented in Table 5.8 and Table 5.9, respectively. The following inferences are drawn from the results.

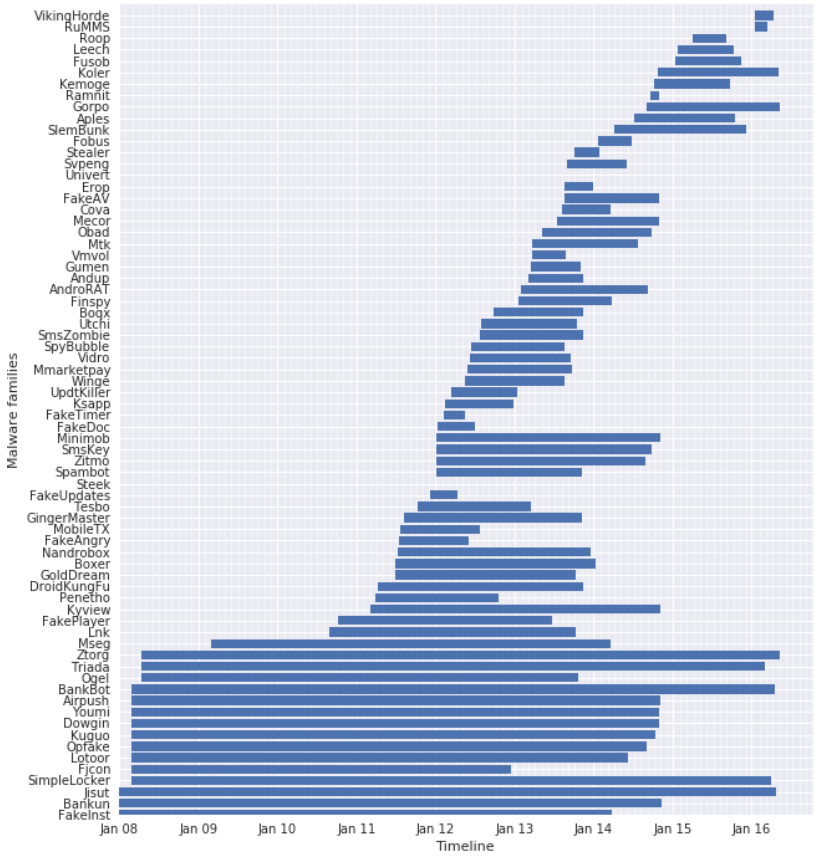
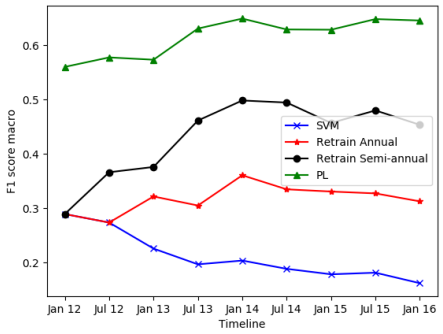


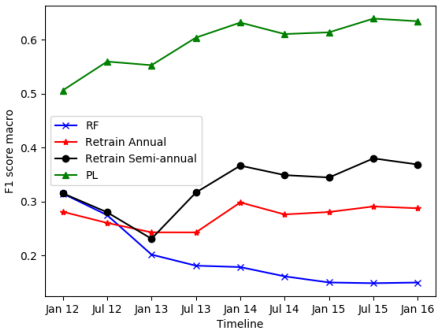
FIGURE 5.2: Bar chart indicating the time period where malware samples from each family exist within

TABLE 5.8: `Drebin` streaming classes efficiency

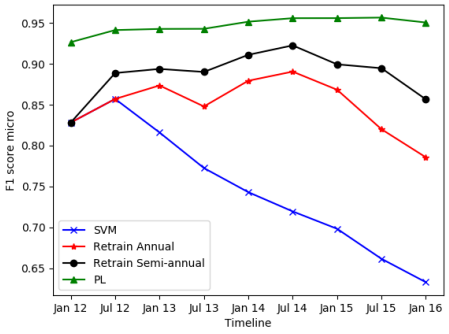
Year (FH/SH)	**# of samples**	**Drebin_ Once**		**Drebin_ Annual**		**Drebin_ Semi_Annual**		**Drebin_ Progressive**	
		Train time	**Test time**	**Train time**	**Test time**	**Train time**	**Test time**	**Train time**	**Test time**
2012(FH)	2,097	0.58	0.43	0.55	0.41	0.76	0.58	30.71	2.04
2012(SH)	3,686	-	0.75	-	0.69	1.46	1.08	51.04	3.36
2013(FH)	6,328	-	0.73	1.78	0.67	2.60	1.10	46.31	3.06
2013(SH)	8,729	-	1.48	-	1.34	3.59	2.12	89.19	6.08
2014(FH)	13,316	-	1.27	4.61	1.19	6.45	1.80	75.12	5.00
2014(SH)	17,048	-	1.15	-	1.07	7.17	1.52	71.33	4.77
2015(FH)	20,492	-	0.20	7.85	0.18	10.55	0.21	12.14	0.81
2015(SH)	21,077	-	0.40	-	0.39	12.44	0.49	25.08	2.53
2016(FH)	22,281	-	0.36	9.86	0.53	10.56	0.44	20.80	1.69



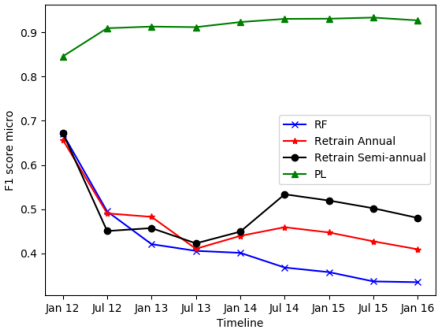
(A) `Drebin` batch learning vs progressive learning average macro F1 score



(B) `CSBD` batch learning vs progressive learning average macro F1 score



(C) `Drebin` batch learning vs progressive learning average micro F1 score



(D) `CSBD` batch learning vs progressive learning average micro F1 score

FIGURE 5.3: Average macro F1 score for (a) `Drebin`, (b) `CSBD` and Average micro F1 score (c) `Drebin` and (d) `CSBD`

TABLE 5.9: `CSBD` streaming classes efficiency

Year (FH/SH)	**# of samples**	**CSBD_ Once**		**CSBD_ Annual**		**CSBD_ Semi_Annual**		**CSBD_ Progressive**	
		Train time	**Test time**	**Train time**	**Test time**	**Train time**	**Test time**	**Train time**	**Test time**
2012(FH)	2,097	3.80	230.26	6.94	228.26	3.62	216.23	36.06	10.92
2012(SH)	3,686	-	392.23	-	386.50	8.23	412.37	64.28	22.79
2013(FH)	6,328	-	377.10	15.61	451.43	15.65	436.67	64.86	38.26
2013(SH)	8,729	-	714.16	-	866.76	24.74	854.77	108.53	63.92
2014(FH)	13,316	-	585.20	42.22	841.98	41.86	838.53	93.20	83.12
2014(SH)	17,048	-	553.32	-	786.73	62.40	852.94	103.59	120.85
2015(FH)	20,492	-	87.59	93.12	138.60	80.22	140.27	13.35	87.79
2015(SH)	21,077	-	177.99	-	305.62	69.87	285.88	28.10	91.95
2016(FH)	22,281	-	267.22	84.67	266.59	76.51	262.81	30.53	104.97

From Figure 5.3, it is clear for both `Drebin` and `CSBD` that when their models are only trained once on malware samples before year 2012, their detection accuracy degrades rapidly over time. This outcome is expected as Figure 5.2 shows that more than half of the malware families in this dataset start to stream in after 2012. Since the models based on the typical multiclass classification algorithms are unable to classify the samples belonging to new classes of malware families that they encounter, they are likely to achieve sub-optimal results. We further observed that in terms of retraining, the detection performance is better when the models are retrained more frequently. This is inline with the observation in the above experiment on streaming samples. As illustrated in Figure 5.2 some new malware families may happen to surface in the first half of year while others in the second half. The semi-annually retrained variants have the advantage over the annually retrained variants when malware samples from new class of malware families surfaced in the first half of the year. Therefore, retraining indeed allow the models to adapt to the new malware family classes encountered by the models over time and the best case scenario is to retrain or update the model as and when a sample from new malware family has surfaced (i.e, progressive learning).

Again from Figure 5.3, we can clearly see that both progressive learning variants significantly out perform their batch learning counterpart with and without retraining. This is due to the capability of the progressive learner to instantaneously update itself to cater to the malware samples that belong new malware family class(es). Whereas for the retrain variants the detection performance is dependent on when the samples belonging to new class of malware family occur and when the retraining will be performed. For example, the retrain models will perform poorly if large volume of samples belonging to new class(es) are encountered right after retraining, then the models will not be able to classify them appropriately at least until the next retraining interval. In addition, the consistent F1 scores, together with the fact that several malware families exist for an extended period of time as depicted in Figure 5.2, demonstrate that learning additional classes does not impair the progressive learner's ability to identify the previously learnt classes.

In terms of efficiency, it is evident from Table 5.8 and Table 5.9 that retraining of the batch models is an incrementally costly operation, not only due to the increasing number of samples added to the training set, but also the increasing complexity due to more number of classes. We also notice that when the number of training

samples is small, the training duration of the progressive learning variants is longer compared to other variants. However, over time as the number of samples increases the training duration of the retraining variants increases continuously. On the other hand, the training duration of the progressive learning variants does not increase over time, but instead varies with the number of samples in the given interval. Testing duration wise, the progressive learning model is slightly slower than SVM but much faster than RF.

Based on the findings above, we conclude that the inability to handle new classes severely limits the competency of the batch learning models in providing real-world malware familial classification solution. Retraining can improve the accuracy of the batch learning models, but the compounding resource requirement is a great cause for concern. Progressive learning provides a high effective and efficient alternative to retraining of the batch learning models.

5.5 Threats to Validity

As any empirical study, our evaluation is subject to multiple threats to validity. In this section we discuss the main threats to validity that can affect the study we have performed.

In terms of threats to external validity, the dataset of over 80K apps used in our study is modest in comparison to the millions of apps available. Therefore, our findings may not hold for other dataset. However, to mitigate this risk, our dataset are collected from multiple sources and across different categories.

One major threat to internal validity concerns the correctness of our implementation of ML based Android malware detection approaches we used as example in our study. To mitigate this threat, we have shown through experiment that our implementation can produce results close the original work and, in addition, we make our implementation publicly available.

Lastly, with regards to threats to construct validity, standard evaluation metrics such as precision, recall, F-measure and CER were used for binary malware detection experiments. Furthermore, macro and micro averaging were used for multiclass

malware familial classification. As such, we believe that the threats to construct validity is minimum.

5.6 Related Work

Empirical studies like the one presented in this work, are essential to ensure that research community are following proper research direction. Roy et el. [91] identify several common challenges, such as proper evaluation and design decision, faced by ML approaches for Android malware detection. From there they study the impact of these challenges through experiments by varying the corresponding parameters. Allix et al. [108] investigate whether "in the lab" Android malware detection validation translates to reliable indications of malware detector's performance in real-world setting, where the number of benign apps greatly overshadow the number of malware and the difficulty of establishing a high quality ground truth is high. Our work is different from them with a focus on the effects of concept drift and extensively study both the impact and possible solutions.

The topic on Android malware evolution are discussed in some of the existing literature. Zhou et al. [50] examines a dataset of 1,260 malware samples belonging to 49 different families to systematically characterize malware infection behavior and study their evolution. They further study the effectiveness of commercial anti-virus software in detecting these malware. Lindorfer et al. [109] present a hybrid Android malware analysis sandbox, namely ANDRUBIS, that generates detailed analysis reports based on both static and dynamic analysis. Based on the analysis results of ANDRUBIS on a dataset of over 1M Android apps where 40% of them are malware, the authors discuss trends in malware behavior observed. Tam et al. [16] present a comprehensive survey on Android malware analysis and detection. They also discuss their effectiveness in face of malware evolution. Our work differs from them by presenting empirical study on challenges and solutions of ML based malware detection approaches in face of concept drift induced by malware evolution.

5.7 Conclusion

We have presented in this chapter, a series of experiments based on two state-of-the-art Android malware detection techniques and demonstrated that their high efficacies do not persist in the real world scenario where history coherency is taken into consideration and thus challenges of concept drift became evident. Beyond that, we have suggested modifications to the approaches on areas where the they did not perform well and showed that the suggestions can significantly improve the performance of the approaches. Finally, we make our implementation of the two state-of-the-art approaches namely, `Drebin` and `CSBD`, publicly available for the research community to advance towards designing successful Android malware detection approach.

Chapter 6

apk2vec: Semi-supervised Multi-view Representation Learning for Profiling Android Applications

6.1 Introduction

As can be observed from the previous chapters, we perform feature engineering each time before we proceed with each of the program analytic tasks. However, the astronomical volume of functionality rich apps, has raised several challenging issues to be addressed through different program analytic tasks. A few significant ones are as follows: (i) app markets are facing difficulties in organizing large volumes of diverse apps to allow convenient and systematic browsing by the users, (ii) due to the rapid growth rate in app volumes, it is becoming increasingly tough for markets to recommend up-to-date and meaningful apps that matches users' search queries, and (iii) with a significant number of plagiarists and malware authors hidden among app developers, these markets have been plagued with app clones and malicious apps. One could observe that a systematic and deep understanding of apps' behaviors is essential to solve the aforementioned issues. Building a holistic and high-quality behavior profiles of apps could help in addressing these issues and avoid performing feature engineering for each individual task.

Past studies [18–26, 99] have demonstrated that in comparison to traditional feature representations of programs (e.g., counts of system-calls, APIs used etc.), graph representations (e.g., CFGs, call graphs, etc.) are ideally suited for app profiling, as the latter retain program semantics well, even when the apps are obfuscated. Reinforcing this fact, many recent works achieved excellent results using graph representations along with Machine Learning (ML) techniques on a plethora of program analytics tasks such as malware detection [18, 20, 21, 26, 99], familial classification [25], clone detection [19, 110], library detection [75] etc. In effect, these works cast their respective program analytics task as a graph analytics task and apply existing graph mining techniques [21] to solve them.

Typically, the aforementioned ML techniques work on vectorial representations (*aka* embeddings) of the graphs. Hence, arguably, one of the most important factors that determines the efficacy of these downstream analytics tasks is the quality of such embeddings.

Besides the choice of graph representations, another pivotal factor that influences the aforementioned tasks are the features that could be extracted from them. In the case of app analytics, the most prominent features in recent literature include API/system-call sequences observed [99], permissions [43] and information source/sinks used [111], etc. Evidently, each of these feature sets provides a different semantic *perspective* (interchangeably referred as *view*) of the apps' behavior with different inherent strengths and limitations. As revealed by existing works [18, 43], capturing multiple semantic views with different modalities would help to improve the accuracy of downstream tasks, significantly. Furthermore, any form of labeling information (e.g., malware family label, app category label, etc.) could be of significant help in building semantically richer and more comprehensive representations.

Towards catering the above-mentioned applications, in this work, we propose a Representation Learning (RL) technique to build data-driven, compact and versatile behavior profiles of apps. Based on the above observations, challenges C4.1-C4.4 have to be addressed to obtain such a profile.

Our approach. Driven by these motivations, we develop a static analysis based RL framework named `apk2vec` which incorporates semi-supervised and multi-view learning paradigm to build high-quality data-driven profiles of Android apps.

`apk2vec` has two main phases: (1) *a static analysis phase* in which a given *apk* file is disassembled and analyzed to extract three different dependency graphs (DGs), each representing a distinct semantic view and, (2) *an embedding phase* in which a neural network integrates the information from these three DGs and label information (if available) to learn one succinct embedding for the *apk*. To this end, `apk2vec` combined and extends several state-of-the-art RL ideas such as multimodal (*aka* multi-view) RL, semi-supervised neural embedding and feature hashing.

More specifically, `apk2vec` addresses the above-mentioned challenges based on its following characteristics:

- **Data-driven embedding:** Instead of the traditional graph kernel methods, `apk2vec` is build upon a varaint of the skipgram neural network [112, 113] that automatically learns features from large corpus of graph data to produce high quality dense embeddings. This in effect addresses challenge C4.1.
- **Semi-supervised task-agnostic embedding:** `apk2vec`'s neural network facilitates the use of apks' class labels (can be single or multiple labels per sample) where available to build better app profiles. However, unlike fully supervised embeddings, these embeddings are still task-agnostics and hence can be used for a variety of downstream tasks. This helps addressing challenge C4.2.
- **Hash embedding:** Recently, Svenstrup et al [114] proposed a scalable feature hashing based word embedding model which required much lesser number of trainable parameters than conventional RL models. Besides improving the memory efficiency, hash embedding technique also facilitates learning embeddings when instances stream over time. Inspired by this idea, in `apk2vec`, we develop an efficient hash embedding model for graph/subgraph embedding, thus addressing challenge C4.3.
- **Multi-view embedding:** `apk2vec`'s neural network facilitates multimodal RL through a novel learning strategy (Section 6.4.3). This helps to integrate three different DGs that are obtained from a given *apk* file from different views in a systematic and non-linear manner to produce one common embedding. Thus addressing challenge C4.4.

Experiments. To evaluate our approach, we perform a series of experiments on various app analytics tasks (including supervised, semi-supervised and unsupervised learning tasks), using a dataset of more than 42,000 real-world Android apps.

The results show that our semi-supervised multimodal embeddings can achieve significant improvements in terms of accuracies over unsupervised/unimodal RL approaches and graph kernel methods while maintaining comparable efficiency. The improvements in prediction accuracies range from 1.74% to 5.93%.

In summary, we make the following contributions:

- We propose `apk2vec`, a static analysis based data-driven semi-supervised multi-view graph embedding framework, to build task-agnostic profiles for Android apps (Section 6.4). To the best of our knowledge, this is the first app profiling framework that has three aforementioned unique characteristics.
- We propose a novel variant of the skipgram model by introducing a view-specific negative sampling which facilitates integrating information from different views in a non-linear manner to obtain multi-view embeddings.
- We extend the feature hashing based word embedding model to learn multi-view graph/subgraph embeddings. Hash embeddings improve `apk2vec`'s overall efficiency and support online RL.
- We make an efficient implementation of `apk2vec` and the profiles of all the apps used in this work publicly available at `https://sites.google.com/view/apk2vec/home`.

6.2 Problem Statement

Given a set of *apk*s $\mathbb{A} = \{a_1, a_2, ...\}$, a set of corresponding labels $\mathbb{L} = \{l_1, l_2, ...\}$ (some of which may be empty i.e., $\forall l_i \in \mathbb{L}, |l_i| \geq 0$) for each app in $\mathbb{A}$ and a positive integer δ (i.e., embedding size), we intend to learn δ-dimensional distributed representations for every *apk* $a_i \in \mathbb{A}$. The matrix representations of all *apk*s is denoted as $\Phi^{\mathbb{A}} \in \mathbb{R}^{|\mathbb{A}| \times \delta}$.

More specifically, $a_i \in \mathbb{A}$ can be represented as $a_i = (G_i^v)$ where $v \in \{A, P, S\}$ and G_i^A, G_i^P, G_i^S denote its API Dependency Graph (ADG), Permission Dependency Graph ($P_m DG$), and information Source & sink Dependency Graph (SDG), respectively (refer to Section 6.4.2 for details on constructing these DGs). Furthermore, a DG can be represented as $G_i^v = (N_i^v, E_i^v, \lambda^v)$, where N_i^v is the set of nodes

and $E_i^v \subseteq N_i^v \times N_i^v$ is the set of edges in G_i^v. A labeling function $\lambda^v : N_i^v \rightarrow L^v$ assigns a label to every node in N_i^v from alphabet set L^v.

Given $G^v = (N^v, E^v, \lambda^v)$ and $sg^v = (N_{sg}^v, E_{sg}^v, \lambda_{sg}^v)$. sg^v is a subgraph of G iff there exists an injective mapping $\mu : N_{sg}^v \rightarrow N^v$ such that $(n_1, n_2) \in E_{sg}^v$ iff $(\mu(n_1), \mu(n_2)) \in E^v$. In this work, by subgraph, we strictly refer to a specific class of subgraphs, namely, rooted subgraphs. In a given graph G^v, a rooted subgraph of degree d around node $n \in N^v$ encompasses all the nodes (and corresponding edges) that are reachable in d hops from n.

6.3 Background & Related Work

Our goal is to build a succinct and versatile behavior profiles of *apk* files in a scalable manner. To this end, we develop a novel *apk* embedding framework which integrates several RL ideas such as document, graph and feature hashing embedding models. We review the related background from these areas in the following.

6.3.1 Skipgram Word and Document Embedding Model

The state-of-the-art word embedding model, `word2vec` [112], produces word embeddings that are capable of encoding the syntactic and semantic regularities. To embed words, `word2vec` uses a simple feed-forward neural network architecture called *skipgram*. It exploits the notion of context such that, given a sequence of words $\{w_1, w_2, ..., w_t, ..., w_T\}$, the target word w_t whose representation has to be learnt and the length of the context window c, the objective of skipgram model is to maximize the following log-likelihood:

$$\sum_{t=1}^{|\mathcal{T}|} \log \Pr(w_{t-c}, ..., w_{t+c}|w_t) \approx \sum_{t=1}^{|\mathcal{T}|} \log \prod_{-c \leq j \leq c, j \neq 0} \Pr(w_{t+j}|w_t) \tag{6.1}$$

where $w_{t-c}, ..., w_{t+c}$ are the context words and $\mathcal{T}$ is the vocabulary of all the words. Here, the context and target words are assumed to be independent. Furthermore, the term $\Pr(w_{t+j}|w_t)$ is defined as: $\frac{e^{(\vec{w_t} \cdot \vec{w'}_{t+j})}}{\sum_{w \in \mathcal{T}} e^{(\vec{w_t} \cdot \vec{w})}}$ where $\vec{w}$ and $\vec{w'}$ are the input and output embeddings of word w, respectively. In the cases where the vocabulary $\mathcal{T}$ is

very large, the posterior probability in eq.(1) could be learnt in an efficient manner using the so-called *negative sampling* technique.

Negative sampling. In each iteration, instead of considering all words in $\mathcal{T}$ a small subset of words that do not appear in the target word's context are selected at random to updates their embeddings. Training this way ensures the following: *if a word w_t appears in the context of another word w_c, then the embedding of w_t is closer to that of w_c compared to any other randomly chosen word from $\mathcal{T}$.* Once skipgram training converges, semantically similar words are mapped to closer positions in the embedding space which is evident that the learnt embeddings is capable of encoding the word semantics.

Le and Mikolov's `doc2vec` [115] extends the skipgram model in a straight forward manner to learn representations of arbitrary length word sequences which include sentences, paragraphs and whole documents. Given a set of documents $\mathbb{D} = \{d_1, d_2, ...\}$ and a set of words $c(d_i) = \{w_1, w_2, ...\}$ sampled from document $d_i \in \mathbb{D}$, `doc2vec` skipgram learns a δ dimensional embeddings of the document d_i and each word $w_j \in c(d_i)$. The model works by considering a word $w_j \in c(d_i)$ to be occurring in the context of document d_i and tries to maximize the following log likelihood:

$$\sum_{j=1}^{|c(d_i)|} \log \Pr\ (w_j|d_i) \tag{6.2}$$

where the probability $\Pr(w_j|d_i)$ is defined as:

$$\frac{e^{(\vec{d}\cdot\vec{w_j})}}{\sum_{w\in\mathcal{T}} e^{(\vec{d}\cdot\vec{w})}} \tag{6.3}$$

Here, $\mathcal{T}$ is the vocabulary of all the words across all documents in $\mathbb{D}$. Understandably, as a straightforward extension of the `word2vec` skipgram model, `doc2vec` could also be trained efficiently with the negative sampling technique.

Model parameters. From the explanations above, it is evident that total number of trainable parameters of the word and the document embedding skipgram models would be $2|\mathcal{T}|\delta$ and $\delta(|\mathbb{D}| + |\mathcal{T}|)$, respectively.

6.3.2 Hash Embedding Model

Though skipgram emerged as a hugely successful embedding model, it faces scalability issues when the vocabulary $\mathcal{T}$ is very large. Also, its architecture does not support embedding of new words (aka *new tokens*) which emerge in an online setting. To address these issues, Svenstrup et al [114] proposed a feature hashing based word embedding model. This model involves the following steps:
(1) A token to id mapping function, $\mathcal{F} : \mathcal{T} \rightarrow \{1, ..., K\}$ and k hash functions of the form $\mathcal{H}_i : \{1, ..., K\} \rightarrow \{1, ..., B\}, i \in [1, k]$ are defined (B is the number of hash buckets and $B << K$).
(2) The following arrays are initialized: $\Phi^{B \times \delta}$: a pool of B component vectors which are intended to be shared by all words in $\mathcal{T}$, and $p^{K \times k}$: contains the importance of each component vector for each word.
(3) Given a word $w \in \mathcal{T}$, hash functions $\mathcal{H}_1, ...\mathcal{H}_k$ are used to choose k component vectors from the shared pool Φ.
(4) The component vectors from step (3) are combined as a weighted sum to obtain the hash embedding of the word w: $\vec{w} = \sum_{w,i=1}^{i=k} p_w^i \mathcal{H}_i(w)$.
(5) With hash embeddings of target and context words thus obtained, skipgram model could be used to train for eq. (1). However, unlike regular skipgram which considers Φ alone as a set of trainable parameters, one could train p as well.

Thus the model reduces the number of trainable parameters from $2K\delta$ to $2(B\delta + Kk)$, which helps reducing the pretraining time and memory requirements. The effect of collisions from K to B could be minimized by having more than one hash function and this helps in maintaining accuracies on-par with `word2vec`. Besides, when new words arrive over time, a function like MD5 or SHA1 could be used in place of $\mathcal{F}$ to hash them to a fixed set of integers in range $[1, K]$.

6.3.3 Graph Embedding Models

Analogous to the `word2vec` model, `graph2vec` [113] considers simple node labeled graphs such as CFGs of arbitrary size as documents and the rooted subgraphs around every node in those graph as words that compose the document. The intuition is that different subgraphs compose graphs in a similar way that different words compose documents. In this way, `graph2vec` could be perceived as an RL

variant of Weisfeiler-Lehman kernel (`WLK`) which counts the number of common rooted subgraphs across a pair of graphs to estimate their similarity.

Given a dataset of graphs $\mathbb{G} = \{G_1, G_2, ...\}$, `graph2vec` extends the skipgram model explained in Section 6.3.1 to learn embeddings of each graph. Let $G_i \in \mathbb{G}$ be denoted as (N_i, E_i, λ) and the set of all rooted subgraphs around every node $n \in N_i$ (up to a certain degree D) be denoted as $c(G_i)$. `graph2vec` aims to learn a δ dimensional embeddings of the graph G_i and each subgraph sg_j sampled from $c(G_i)$ i.e., $\vec{G_i}, \vec{sg_j} \in \mathbb{R}^\delta$, respectively by maximizing the following log likelihood: $\sum_{j=1}^{|c(G_i)|} \log \Pr\ (sg_j|G_i)$, where the probability $\Pr\ (sg_j|G_i)$ is defined as: $\frac{e^{(\vec{G} \cdot \vec{sg_j})}}{\sum_{w \in \mathcal{T}} e^{(\vec{G} \cdot \vec{sg})}}$. Here, $\mathcal{T}$ is the vocabulary of all the subgraphs across all graphs in $\mathbb{G}$. The number of trainable parameters of this model will be $\delta(|\mathbb{G}| + |\mathcal{T}|)$.

Similar to `graph2vec`, many recent approaches such as `sub2vec` [116], `GE-FSG` [117] and Anonymous Walk Embeddings (AWE) [118] have adopted skipgram architecture to learn unsupervised graph embeddings. The fundamental difference among them is the type of graph substructure that they consider as a graph's context. For instance, `sub2vec` considers nodes, `GE-FSG` considers frequent subgraphs (FSGs) and AWE considers walks that exist in graphs as their contexts, respectively.

6.3.4 Semi-supervised Embedding Model

Recently, Pan et al [119] extended the skipgram model to learn embedding of nodes in Heterogeneous Information Networks (HINs) with semi-supervision. More specifically, their variant of skipgram model facilitates the use of class labels for a subset of samples while learning embeddings. For instance, when l_i, the class label of a document d_i is available, the `doc2vec` model could maximize the following log likelihood:

$$\beta \sum_{j=1}^{|c(d_i)|} \log \Pr\ (w_j|d_i) + (1-\beta) \sum_{j=1}^{|c(d_i)|} \log \Pr\ (w_j|l_i) \tag{6.4}$$

to include the supervision signal available from l_i along with the contents of the document made available through $c(d_i)$. Here, β is the weight that balances the importance of the two components in document embedding. Pan et al. empirically prove that embeddings with this form of semi-supervision significantly improves the accuracy of downstream tasks.

In summary, all above-mentioned embedding models provide different advantages for RL. Drawing from the strengths of each model, we propose a novel data-driven graph embedding framework for app behaviour profiling. Our framework possess three critical factors which provide the ability for building holistic app behavior profiles, namely: (i) multi-view RL, (ii) semi-supervised RL, and (iii) feature hashing based RL. We discuss in principle and demonstrate through experiments that our framework possess all these strengths and can cater to various downstream tasks.

6.4 Methodology

In this section, we explain the `apk2vec` app profiling framework. We first present an overview of the framework which encompasses two phases, subsequently, we discuss the details of each component in the following subsections.

6.4.1 Framework Overview

As depicted in Figure 6.1, the workflow of `apk2vec` can be divided into two main phases, namely, the *static analysis phase* and the *embedding phase*. The static analysis phase encompasses of static program analysis procedures which extracts DGs for *apk* files. The subsequent embedding phase encompasses RL techniques that transform these DGs into *apk* profiles.

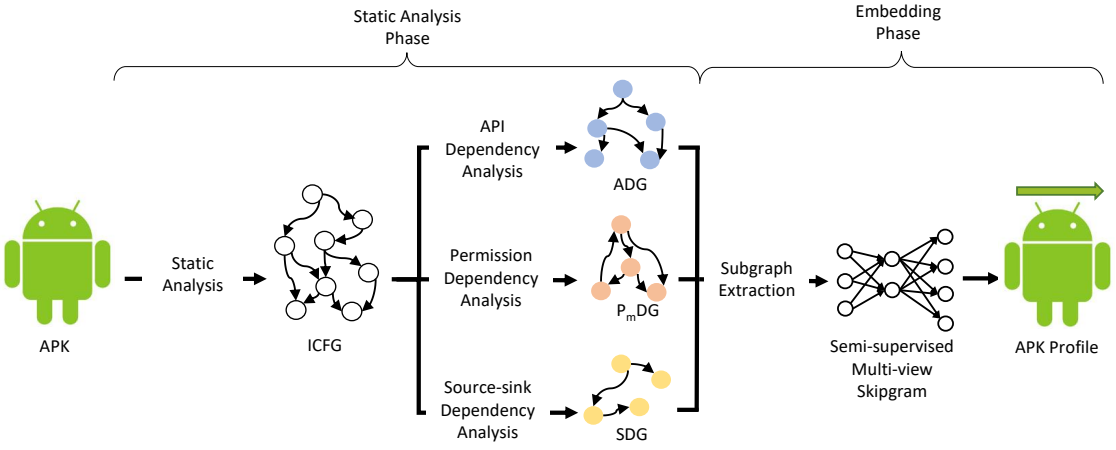


FIGURE 6.1: `apk2vec`: Framework overview

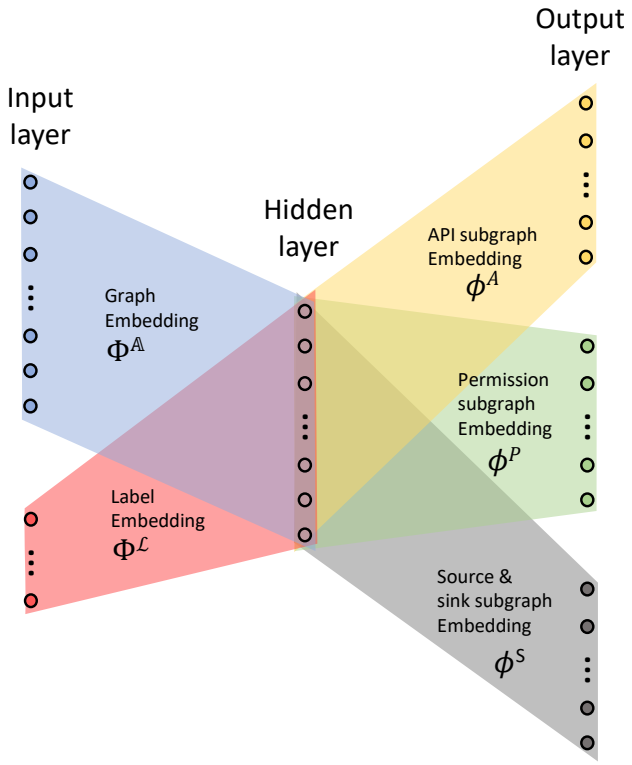


FIGURE 6.2: Semi-supervised multi-view skpigram

Static analysis phase. In this phase, we begin by disassembling the *apk*s for a given dataset and perform static analysis to construct their interprocedural CFG (ICFG). We then perform further analysis to abstract each of the ICFGs into three semantically different DGs, namely, ADG, P_mDG and SGD. Each of them represents a unique semantic view of the app's behaviors with distinct modalities. Detailed procedure of constructing these DGs is presented in Section 6.4.2.

Embedding phase. After constructing the DGs for all the *apk*s in the dataset, we extract the rooted subgraphs around every node in the DGs to facilitate the learning of *apk* embeddings. Upon completion of the rooted subgraphs extraction, we train the semi-supervised multi-view skpigram neural network with them. The detailed procedures in this phase are discussed in Section 6.4.3.

6.4.2 Static Analysis Phase

ICFG construction. For a given *apk*, we first perform static control-flow analysis to builds its ICFG. The ICFG is chosen over other form of program representation graphs (e.g., DFGs, call graphs) due to its fine-grained representation of control

flow sequence, which allows us to capture finer semantic details of the *apk* which is necessary to build a comprehensive *apk* profile. Formally, $ICFG = (N, E)$ for an *apk* a is a directed graph where each node $bb \in N$ denotes a basic block[1] of a method m in a, and each edge $e(bb_1, bb_2) \in E$ denotes either an intra-procedural control-flow or a calling relationship from bb_1 to bb_2 and $E \subseteq N \times N$.

Abstraction into multiple views. To encode the *apk* with richer semantics from different modalities, we abstract it with three Android platform specific analysis, namely API sequences, Android permissions, and information sources & sinks to construct the three DGs, respectively. The abstraction process is described below.

To obtain the ADG from a given ICFG, we remove all nodes that do not access security sensitive Android APIs. This will leave us with a subset of sensitive nodes from the perspective of API usages, say $N^{\mathcal{A}} \subseteq N$. Subsequently, we add an edge between a pair of nodes $n_1, n_2 \in N^{\mathcal{A}}$ iff there exist a path between them in ICFG. This yields the ADG, $G^{\mathcal{A}}$ which could be formally represented as a three tuple $G^{\mathcal{A}} = (N^{\mathcal{A}}, E^{\mathcal{A}}, \lambda^{\mathcal{A}})$, where $\lambda^{\mathcal{A}} : N^{\mathcal{A}} \rightarrow L^{\mathcal{A}}$ is a labeling function that assigns a security sensitive API as a node label to every node in $N^{\mathcal{A}}$ from a set of alphabets $L^{\mathcal{A}}$. We refer to existing work [99] for the list of security sensitive APIs. Similarly, we use works such as PScout [120] and SUSI [111] which maps APIs to Android permissions and information source/sinks to obtain set of node labels $L^{\mathcal{P}}$ and $L^{\mathcal{S}}$, respectively. Subsequently, adopting the process mentioned above using them we abstract the ICFG into $P_m DG$ and SDG using $L^{\mathcal{P}}$ and $L^{\mathcal{S}}$, respectively.

6.4.3 Embedding Phase

In the embedding phase, the objective is to train the skipgram model with the DGs and class labels that belongs to a set of apks and obtain the behavior profile for each of the apk. To this end, we develop a novel variant of the skipgram model which facilitates incorporating the three following learning paradigms: semi-supervised RL, multi-view RL and feature hashing.

Network architecture. Figure 6.2 depicts the architecture of the neural network used in `apk2vec`'s RL process. The network consists of two shared input layers and three shared output layers (one for each view). The goal of the first input

[1] A basic block is a sequence of instructions in a method with only one entry point and one exit point which represents the largest piece of the program that is always executed altogether.

layer ($\Phi^{\mathbb{A}}$) is to perform multi-view RL. More precisely, given an *apk* id a_i in the first layer, the network intends to predict the API, permission and source-sink subgraphs that appear in a_i's context, in each of the output layers. Whereas the function of the second input layer ($\Phi^{\mathcal{L}}$) is to facilitate semi-supervised RL. More specifically, given a_i's class label as input in the second layer, the network intends to predict subgraphs of all three views that occur in a_i's context in each of the output layers. Thus, the network forces API, permission and source-sink subgraphs that frequently co-occur with same class labels to have similar embeddings. For instance, given a malware family label $\mathfrak{f}$, subgraphs that characterize $\mathfrak{f}$'s behaviors would end up having similar embeddings. This in turn would influence *apk*s that belong to $\mathfrak{f}$ to have similar embeddings.

Hash embeddings. Considering the real-world scenario where Android apps are constantly evolving (i.e., APIs/permissions being added or deprecated) and new apps emerge rapidly over time, clearly the vocabulary of subgraphs (across all DGs) would grow as well. Regular skipgram models could not handle such dynamic vocabulary and as mentioned in Section 6.3.2, hash embeddings could be used effectively to address this. Note that in our framework, the vocabulary of subgraphs is only present in the output layer. Hence, in `apk2vec`, hash embeddings are used only in the three output layers (ϕ^A, ϕ^P and ϕ^S) and the two input layers ($\Phi^{\mathbb{A}}$ and $\Phi^{\mathcal{L}}$) uses regular embeddings.

The process through with our skipgram model is trained is explained through Algorithm 3.

6.4.4 Algorithm: `apk2vec`

The algorithm takes the set of apks along with their corresponding DGs ($\mathbb{A}$), set of their labels ($\mathbb{L}$), maximum degree of rooted subgraphs to be considered (D), embedding size (δ), number of epochs ($\mathcal{E}$), number of hash buckets per view (B^v), number of hash functions (k) and learning rate (α) as inputs and outputs the *apk* embeddings ($\Phi^{\mathbb{A}}$). The major steps of the algorithm are as follows:

1. We begin by randomly initializing the parameters of the model i.e., $\Phi^{\mathbb{A}}$: *apk* embeddings, $\Phi^{\mathcal{L}}$: label embeddings, ϕ^v: embeddings of each hash bucket for each of the three views, and p^v: importance parameters for each of the views

Algorithm 3: APK2VEC $(\mathbb{A}, \mathbb{L}, D, \delta, \mathcal{E}, B^v, k, \alpha)$

Input: $\mathbb{A} = \{a_1, a_2, ...\}$: set of apks such that $a_i = \{G_i^v\}$ for $v \in \{\mathcal{A}, \mathcal{P}, \mathcal{S}\}$
$\mathbb{L} = \{l_1, l_2, ...\}$: Set of labels for each *apk* in $\mathbb{A}$. Note that there may be zero or more labels for an apk. Hence $\forall l_i \in \mathbb{L}, |l_i| \geq 0$. Let the total number of unique labels across $l_i \in \mathbb{L}$ be denoted as $\mathcal{L}$.
D : Maximum degree of rooted subgraphs to be considered for learning embeddings. This will produce a vocabulary of subgraphs in each view, $\mathcal{T}^v = \{sg_1^v, sg_2^v, ...\}$ from all the graphs G_i^v. Let $|\mathcal{T}^v|$ be denoted as K^v.
δ : Number of dimensions (embedding size)
$\mathcal{E}$: Number of epochs
B^v: Number of hash buckets for view v
k: Number of hash functions (maintained same across all views)
α : Learning rate
Output: Matrix of vector representations of apks $\Phi^{\mathbb{A}} \in \mathbb{R}^{|\mathbb{A}| \times \delta}$

1 Initialization: Sample $\Phi^{\mathbb{A}}$ from $\mathbb{R}^{|\mathbb{A}| \times \delta}$, $\Phi^{\mathcal{L}}$ from $\mathbb{R}^{|\mathcal{L}| \times \delta}$, ϕ^v from $\mathbb{R}^{B^v \times k}$, and p^v from $\mathbb{R}^{K^v \times k}$
for $e = \{1, 2, ..., \mathcal{E}\}$ **do**
2 **for** $a_i \in$ SHUFFLE($\mathbb{A}$) **do**
3 **for** $G_i^v \in a_i$ **do**
4 $sg_c :=$ GETSUBGRAPHS(G_i^v, D)
5 $J(\Phi^{\mathbb{A}}, \phi^v, p^v) := -\log \prod_{sg \in sg_c} \frac{e^{(\Phi^{\mathbb{A}}(a_i) \cdot \text{HASHEMB}(sg, \phi^v, p^v, v))}}{\sum_{sg' \in \mathcal{T}^v} e^{(\Phi^{\mathbb{A}}(a_i) \cdot \text{HASHEMB}(sg', \phi^v, p^v, v))}}$
6 $\Phi^{\mathbb{A}} := \Phi^{\mathbb{A}} - \alpha \frac{\partial J}{\partial \Phi^{\mathbb{A}}}$; $\phi^v := \phi^v - \alpha \frac{\partial J}{\partial \phi^v}$; $p^v := p^v - \alpha \frac{\partial J}{\partial p^v}$
7 **for** $l \in l_i$ **do**
8 $J(\Phi^{\mathcal{L}}, \phi^v, p^v) := -\log \prod_{sg \in sg_c} \frac{e^{(\Phi^{\mathcal{L}}(l) \cdot \text{HASHEMB}(sg, \phi^v, p^v, v))}}{\sum_{sg' \in \mathcal{T}^v} e^{(\Phi^{\mathcal{L}}(l) \cdot \text{HASHEMB}(sg', \phi^v, p^v, v))}}$
9 $\Phi^{\mathcal{L}} := \Phi^{\mathcal{L}} - \alpha \frac{\partial J}{\partial \Phi^{\mathcal{L}}}$; $\phi^v := \phi^v - \alpha \frac{\partial J}{\partial \phi^v}$; $p^v := p^v - \alpha \frac{\partial J}{\partial p^v}$
10 **return** $\Phi^{\mathbb{A}}$

(line 1). It is noted that except the *apk* embeddings, all other parameters are discarded when training culminates.

2. For each epoch, we consider each *apk* a_i as the target whose embedding has to be updated. To this end, each of its DG G_i^v is taken and all the rooted subgraphs upto degree D around every node are extracted from the same (line 4). The subgraph extraction process is explained in detail in Section 6.4.5.

3. The set of all such subgraphs sg_c, is perceived as the context of a_i. Once sg_c is obtained, we get their hash embeddings and compute the negative log likelihood of them being similar to the target *apk* a_i's embedding (line 5). The hash embedding computation process is explained in detail in Section 6.4.6.

4. With the loss value thus computed, the parameters that influence the loss are updated (line 6). We propose a novel view-specific negative sampling strategy to train the skipgram and the same is explained in Section 6.4.7.

5. Subsequently, for each of a_i's class labels i.e., $l \in l_i$, we compute the negative log likelihood of their similarity to the context subgraphs in sg_c and update

the parameters that influence the same (lines 7-9). This step amounts to semi-supervised RL as l_i could be empty for some apks.

6. The above mentioned process is repeated for $\mathcal{E}$ epochs and the *apk* embeddings (along with other parameters) are refined.

Finally, when training culminates, *apk* embeddings in $\Phi^{\mathbb{A}}$ are returned (line 10).

6.4.5 Extracting Context Subgraphs

For a given *apk* a_i, extracting rooted subgraphs around each node in each G_i^v and considering them as its context is a fundamental task in our approach. To extract these subgraphs, we follow the well-known Weisfeiler-Lehman relabeling process [121] which lays the basis for `WLK` [121, 122]. The subgraph extraction process is presented formally in Algorithm 4. The algorithm takes the graph G from which the subgraphs have to be extracted and maximum degree to be considered around root node D as inputs and returns the set of all rooted subgraphs in G, S. It begins by initializing S to an empty set (line 2). Then, we intend to extract rooted subgraph of degree d around each node n in the graph. When $d = 0$, no subgraph needs to be extracted and hence the label of node n is returned (line 6). For cases where $d > 0$, we get all the (breadth-first) neighbours of n in $\mathcal{N}_n$ (line 8). Then for each neighbouring node, n', we get its degree $d-1$ subgraph and save the same in a multiset $M_n^{(d)}$ (line 9). Subsequently, we get the degree $d-1$ subgraph around the root node n and concatenate the same with sorted list $M_n^{(d)}$ to obtain the subgraph of degree d around node n, which is denoted as $sg_n^{(d)}$ (line 10). $sg_n^{(d)}$ is then added to the set of all subgraphs (line 11). When all the nodes are processed, rooted subgraphs of degrees $[0, D]$ are collected in S which is returned finally (line 12).

6.4.6 Obtaining hash embeddings

Once we have extracted context subgraphs, we proceed with obtaining their hash embeddings and training the same along the target apk's embedding. Given a subgraph, the process of extracting its hash embedding involves a four step process which is formally presented in Algorithm 5. Following is the explanation of this algorithm:

Algorithm 4: GetSubgraphs (G, D)

```
1  begin
2      S := {} //Initialize with an empty set
3      for n ∈ N do
4          for d ∈ {0, 1, ..., D} do
5              if d = 0 then
6                  sg_n^(d) := λ(n) //node label
7              else
8                  N_n := {n' | (n, n') ∈ E}//neighboring nodes
9                  M_n^(d) := {{GetWLSubgraph(n', G, d − 1) | n' ∈ N_n}} //multiset of rooted subgraphs
                     around neighboring nodes
10                 sg_n^(d) := GetWLSubgraph(n, G, d − 1) ⊕ sort(M_n^(d))
11             S := S ∪ sg_n^(d)
12     return S //set of all rooted subgraphs in G
```

Algorithm 5: HashEmb (sg, ϕ, p, v)

```
1  begin
2      sg_id := F^v(sg) //Token to id mapping function
3      components := (φ(H_1(sg_id)), ..., φ(H_k(sg_id)))^T //shape of components: k × δ
4      weights := (p_1^v(sg_id), p_2^v(sg_id), ..., p_k^v(sg_id))^T //shape of weights: k × 1
5      s⃗g := weights^T · components //1 × k · k × δ
6      return s⃗g
```

1. Given a subgraph sg, we begin by mapping to an integer sg_{id}, using a function $\mathcal{F}^v$ (line 2). When $\mathcal{T}^v$, the vocabulary of all the subgraphs in view v could be obtained ahead of training, a regular dictionary *aka* token-to-id function which maps each subgraph to a unique number in the range $[1, K^v]$ (where $K^v = |\mathcal{T}^v|$) could be used as $\mathcal{F}^v$. In the online learning setting, such a dictionary could not be obtained. Hence, analogous to feature hashing [123], one could use a regular hash function such as MD5 or SHA1 to hash the subgraph to an integer in the predetermined range $[1, K^v]$ (here, an arbitrarily large value of K^v is chosen to avoid collisions).

2. sg_{id} is then hashed using each of the k hash functions. Each function $\mathcal{H}_i, i \in [1, k]$ maps it to one of the B^v available hash buckets which in turn maps to one of the B^v component embeddings in ϕ^v. Thus we obtain k component embeddings for the given subgraph and save them in *components* (line 3). In other words, *components* contains k δ-dimensional embeddings.

3. Similarly, using sg_{id}, we then lookup the importance parameter for each hash function, $p_i^v(sg_{id}), i \in [1, k]$ and save them in *weights* (line 4). In other words, *weights* contains k importance values.

4. Finally, the hash embedding of the subgraph is obtained by multiplying k δ-dimensional component vectors with k corresponding importance values (line 5).

Once the hash embeddings of the context subgraphs are obtained using the above mentioned process, one could train them along with the target apk's embedding using a learning algorithm such as Stochastic Gradient Decent (SGD).

6.4.7 View-specific Negative Sampling

Similar to other skipgram based embedding models such as `graph2vec` [113], we could efficiently minimize the negative log likelihood in lines 5 and 8 of Algorithm 3. That is, given an *apk* a and a subgraph sg^v which is contained in view v, the regular negative sampling intends to maximize the similarity between their embeddings. Besides, it chooses η subgraphs as negative samples i.e., that do not occur in the context of a and minimizes the similarity of a and these negative samples. This could be formally presented as follows,

$$\Pr(sg^v|a) = \sigma(\vec{a}^T \cdot \vec{sg^v}) \prod_{j=1}^{\eta} \mathbb{E}_{sg_j \sim \Pr_n(\mathcal{T})} \sigma(-\vec{a}^T \cdot \vec{sg_j}) \tag{6.5}$$

where, $\mathcal{T} = \bigcup_v \mathcal{T}^v$ is union of vocabularies across all views and $\mathbb{E}$ is expectation of choosing a subgraph sg_j from the smoothed distribution of subgraphs $\Pr_n$ across all the three views.

In other words, eq. (3) moves $\vec{a}$ closer to $\vec{sg^v}$ as it occurs in a's context and also moves $\vec{a}$ farther away from $\vec{sg_j}$ (which may not belong to view v) as it does not occur in a's context.

However, in our multi-view embedding scenario, the distribution of subgraphs is not similar across all views. For instance, in our experiments reported in Section 6.5, the API view produces millions of subgraphs, where as the permission and source-sink view produce only thousands. Hence, eq. (3) which ignores the view-specific probability of subgraph occurrences is not suitable in this scenario. Therefore, we propose a novel view-specific negative sampling strategy as described by the

equation below:

$$\Pr(sg^v|a) = \sigma(\vec{a}^T \cdot \vec{sg^v}) \prod_{j=1}^{\eta} \mathbb{E}_{sg_j^v \sim \Pr_n(\mathcal{T}^v)} \sigma(-\vec{a}^T \cdot \vec{sg_j^v}) \tag{6.6}$$

In simpler terms, eq. (4) moves $\vec{a}$ closer to $\vec{sg^v}$ as it occurs in a's context and also moves $\vec{a}$ farther away from $\vec{sg_j^v}$ (which also belongs to view v) as it does not occur its a's context.

6.4.8 Model Dynamics

The trainable parameters of our model are $\Phi^{\mathbb{A}}, \Phi^{\mathcal{L}}, \phi^v$, and p^v. Recall, $\Phi^{\mathbb{A}}$ and $\Phi^{\mathcal{L}}$ are regular embeddings as they are in the input layers and ϕ^vs are hash embeddings. Also, the total number of tokens in the input and output layers would be $|\mathbb{A}| + |\mathcal{L}|$ and $\sum_v K^v$, respectively. Hashing (which is applicable only to ϕ^v) reduces the number of parameters in the output layer from $\sum_v K^v$ to $K^v k + kB^v$ where $B^v << K^v$ (typically, we set $k = [2, 4]$ and $K^v > B^v \cdot 100$).

From the explanations above, it is evident that the computational overhead of using hash embeddings instead of standard embeddings is mainly contributed by the embedding lookup step. More precisely, a multiplication of a $1 \times k$ matrix (obtained from p^v) with a $k \times \delta$ matrix (obtained from ϕ^v) is required instead of a regular matrix lookup to get $1 \times \delta$ subgraph embedding. When using small values of k, the computational overhead is therefore negligible. In our experiments, hash embeddings are marginally slower to train than standard embeddings on datasets with small vocabularies.

6.5 Evaluation

We evaluate the efficacy of `apk2vec`'s embeddings with several tasks involving various learning paradigms that include supervised learning (batch and online), unsupervised learning and link prediction. The evaluation is carried out on five different datasets involving a total of 42,542 Android apps. In this section, we first present the experimental design aspects, such as research questions addressed,

datasets and tasks chosen pertaining to the evaluation. Subsequently, the results and relevant discussions are presented.

Research questions. Through our evaluations, we intend to address the following questions:

- How accurate do `apk2vec`'s embeddings perform on various app analytics tasks and how do they compare to state-of-the-art approaches?
- Do multi-view profiles offer better accuracies than single-view profiles?
- Does semi-supervised RL help improving the accuracy of app profiles?
- How does `apk2vec`'s hyperparameters affect its accuracy and efficiency?

Evaluation setup. All the experiments were conducted on a server with 40 CPU cores (Intel Xeon(R) E5-2640 2.40GHz), 6 NVIDIA Tesla V100 GPU cards with 256 GB RAM running Ubuntu 16.04.

Comparative analysis. To provide a comprehensive evaluation, we compare our approach with four baseline approaches, namely, `WLK` [121], graph2vec [113], `sub2vec`[116] and `GE-FSG` [117]. Refer to Section 6.1 and Section 6.3 for brief explanations on the baselines. The following evaluation-specific details on baselines are noted: (i) Since all baselines are unimodal they are incapable of leveraging all the three DGs to yield one unified *apk* embedding. Hence, to ensure fair comparison, we merge all three DGs into one graph and feed them to these approaches. (ii) `sub2vec` has two variants, namely, `sub2vec_N` (which leverages only neighborhood information for graph embedding) and `sub2vec_S` (which leverages only structural information). Both these variants are included in our evaluations, and (iii) For all baselines except `GE-FSG`, open-source implementations provided by the authors are used. For `GE-FSG`, we reimplemented it by following the process described in their original work. Our reimplementation could be considered faithful as it reproduces the results reported in the original work.

Hyperparameter choices. In terms of `apk2vec`'s hyperparameters, we set the following values: $\mathcal{E} = 100, \delta = 64$, $\alpha = 0.1$ (with decay) and $\eta = 2$. When hash embedding is used $k = 2, B^v = \frac{K^v}{10}$ (for all v). To ensure fair comparison, in all experiments, the hyperparameters of all baseline approaches are maintained same as those of `apk2vec` (e.g., the embedding dimensions of all baselines are set to 64,

TABLE 6.1: Datasets used for evaluations

Task	Data source	# of apps	Avg. nodes			Avg. edges		
			ADG	$P_m DG$	SDG	ADG	$P_m DG$	SDG
Batch malware detection	Malware: [43, 124] Benignware: [101]	19,944 20,000	783.03	131.73	80.12	2604.28	174.64	94.04
Online malware detection	Malware: [43] Benignware: [101]	5,560 5,000	365.18	63.11	37.885	744.99	66.96	34.67
Malware familial clustering	Drebin [43]	5,560	229.82	69.96	43.41	464.73	72.34	44.57
Clone detection	Clone apps [19]	280	674.71	179.09	94.69	1553.29	182.24	76.64
App recommendation	Googleplay [101]	2,318	2168.88	242.87	154.14	5137.47	348.67	150.87

etc.). In all experiments, for datasets where class labels are available, 25% of the labels are used for semi-supervision during embedding (unless otherwise specified).

6.5.1 `apk2vec` vs. state-of-the-art

In the following subsections we intend to evaluate `apk2vec` against the baselines on two classification (i.e., batch and online malware detection), two clustering (i.e., app clone detection and malware familial clustering) and one link prediction (i.e., app recommendation) tasks. The evaluations are performed on the datasets reported in Table 6.1. Notably, the DGs obtained from *apk*s are in general significantly larger compared to the graphs in the benchmark datasets (e.g., datasets used in [117]) and even some large real-world datasets (e.g., used in [122]). Some of the baselines do not scale well to embed such large graphs and they run into *Out of Memory* (OOM) situations.

6.5.1.1 Graph classification

Dataset & experiments. For batch learning based malware detection task, 19,944 malware from two well-known malware datasets [43] and [124] are used. As for the benign apps, 20,000 apps from Google Play [101] and verified with VirusTotal are used. To perform the malware detection, we first obtain profiles of all these apps using `apk2vec`. Subsequently, a SVM classifier is trained with 70% of samples and is evaluated with the remaining 30% samples (classifier hyperparamters are tuned using 5-fold cross-validation). This trial is repeated 5 times and the results are averaged.

For online malware detection task, 5,560 malware from [43] and 5,000 benign apps from Google Play are used. In this experiment, the real-world situation where apps stream in over time is simulated as follows: First, *apk*s are temporally sorted according to their time of release (see [99] for details). Thereafter, the embeddings of first 1,000 *apk*s are used to train an online Passive Agressive (PA) classifier. For the remaining 9,560 *apk*s, their embeddings are obtained from `apk2vec` in an online fashion. These embeddings are fed to the trained PA model for evaluation and classifier update.

Standard metrics such as precision, recall and f-measure are used to evaluate the efficacy for both batch and online settings.

Results & discussions. The batch and online malware detection results are presented in Table 6.2. The following inferences are drawn from the tables.

- In batch learning setting, it is evident from the f-measure that `apk2vec` outperforms all baselines. More specifically, with just 25% labels it is able to outperform the worst and best performing baselines by more than 20% and nearly 2%, respectively. Clearly, this improvement could be attributed to `apk2vec`'s multimodal and semi-supervised embedding capabilities.
- `apk2vec`'s improvements in f-measure are even more prominent in the online learning setting. More specifically, it outperforms the worst and best performing baselines by nearly 35% and more than 5%, respectively. Clearly, this improvement could be attributed to `apk2vec`'s hash embedding capabilities through which it handles dynamically expanding vocabulary of subgraphs.
- Looking at the performances of baselines, one could see all of them perform reasonably better in the batch learning setting than the online setting. This is owing to their inability handle vocabulary expansion which renders their models obsolete over time. Besides, none of them posses multi-view and semi-supervised learning potentials which could explain their overall substandard results.
- Due to poor space complexity, `GE-FSG` [117] is unable to handle the large graphs used in this experiment and went OOM during the FSG extraction process. Hence, its results are not reported in the table.

TABLE 6.2: Malware detection (graph classification) results

Technique	Batch			Online		
	P(%)	**R**(%)	**F**(%)	**P**(%)	**R**(%)	**F**(%)
`apk2vec`	88.07	**90.41**	**89.22**	87.90	89.73	88.81
`WLK`[121]	**88.15**	86.38	87.25	84.13	82.32	83.22
`graph2vec`[113]	76.96	82.48	79.63	82.55	84.21	83.37
`sub2vec_N`[116]	68.31	69.65	68.98	52.20	56.65	54.33
`sub2vec_S`[116]	66.94	68.36	67.64	53.13	54.98	54.04

6.5.1.2 Graph Clustering

Dataset & experiments. In this experiment, we evaluate `apk2vec` on two different graph clustering tasks. Firstly, for the malware familial clustering task, we use the 5,560 apps, belonging to 179 malware families, from the Drebin [43] collection. Malware belonging to the same family are semantically similar as they perform similar attacks. Hence, we obtain the profiles of these apps and cluster them into 179 clusters using k-means algorithm. Profiles of samples belonging to same family are expected end up in the same cluster.

The next task is clone detection which uses 280 apps from Chen et al.'s [19] work. The apps in this dataset belong to 100 clone groups, where each group contains at least two apps that are semantic clones of each other with slight modifications/enhancements. Hence, in this task, we obtain the app profiles and cluster them into 100 clusters using k-means algorithm with the expectation that cloned apps end up in the same cluster.

Adjusted Rand Index (ARI) is used as a metric to determine the clustering accuracy in both these tasks.

Results & discussions. The clustering results are presented in Table 6.3. The following inferences are drawn from the table.

- At the outset, it is clear that `apk2vec` outperforms all the baselines on both these tasks. For familial clustering and clone detection, the improvements over the best performing baselines are 0.07 and 0.01 ARI, respectively.
- Interestingly, unlike malware detection not all the baselines offer agreeable performances in these two tasks. For instance, the ARIs of `sub2vec` and `GE-FSG` are too low to be considered as practically viable solutions. Given this context, `apk2vec`'s performances show that its embeddings generalize well and are task-agnostic.

TABLE 6.3: Malware familial clustering and clone detection (graph clustering) results

Technique	**Familial clustering** (ARI)	**Clone detection** (ARI)
`apk2vec`	**0.5124**	**0.8360**
`WLK`[121]	0.3279	0.7766
`graph2vec`[113]	0.4441	0.8272
`sub2vec_N`[116]	0.0374	0.1801
`sub2vec_S`[116]	0.0945	0.0454
`GE-FSG` [117]	OOM	0.0171

6.5.1.3 Link Prediction

Dataset & experiments. For this task, we constructed an app recommendation dataset consisting of 2,318 apps downloaded from Google Play. We build a recommendation graph $\mathcal{R}$, with these apps as nodes. An edge is placed between a pair of apps in $\mathcal{R}$, if Google Play recommends one of them while viewing the other. With this graph, we follow the procedure mentioned in [125] to cast app recommendation as a link prediction problem. That is, P, a subset of edges (chosen at random) are removed from $\mathcal{R}$, while ensuring that this residual graph $\mathcal{R}'$ remains connected. Now, given a pair of nodes in $\mathcal{R}'$, we predict whether or not an edge exists between them. Here, endpoints of edges in P are considered as positive samples and pairs of nodes with no edge between them in R are considered as negative samples. We perform the experiment for $P = \{10\%, 20\%, 30\%\}$ of total number of edges in $\mathcal{R}$.

Area under the ROC curve (AUC) is used as a metric to quantify the efficacy of link prediction.

Results & discussions. The results of the app recommendation task are presented in Table 6.4 from which the following inferences are drawn.

- For all values of P, `apk2vec` consistently outperforms all the baselines. Also, `apk2vec`'s margin of improvement over baselines is consistent and much higher in this task than graph classification and clustering tasks. For instance, it improves best baseline performances by 3 to 4% across all P values.
- It is noted that link prediction task does not involve any semi-supervision and hence all this improvement could be attributed to `apk2vec`'s multi-view and data-driven embedding capabilities.

TABLE 6.4: App recommendation (link prediction) results

Technique	AUC (P = 10%)	AUC (P = 20%)	AUC (P = 30%)
apk2vec	**0.7187**	**0.7347**	**0.7236**
WLK[121]	0.6643	0.6865	0.6805
graph2vec[113]	0.6830	0.7043	0.6876
sub2vec_N[116]	0.5446	0.5808	0.5403
sub2vec_S[116]	0.5206	0.5632	0.5631

TABLE 6.5: Clone detection results: single- vs. multi-view *apk* profiles

Technique	Views				
	APIs(ARI)	**Perm.**(ARI)	**Src-sink**(ARI)	**concat.**(ARI)	**multi-view**(ARI)
apk2vec	0.8208	0.7855	0.7953	0.8325	**0.8360**
WLK[121]	0.8078	0.7382	0.7479	0.7766	-

In sum, apk2vec consistently offers the best results across all the five tasks reported above. This illustrates that apk2vec's embeddings are truly task-agnostic and capture the app semantics well.

6.5.2 Single- vs. Multi-view Profiles

In this experiment, we intend to evaluate the following: (i) significance of three individual views used in apk2vec, (ii) significance of concatenating app profiles from individual views (i.e. linear combination), and (iii) whether non-linear combination of multiple views is better than (i) and (ii).

Dataset & experiments. To this end, we use the clone detection experiment reported in Section 6.5.1.2. First, we build the app profiles (i.e., 64-dimensional embedding) with individual views (i.e., only one output layer is used in skipgram). Clone detection is then performed with each view's profile. Also, we concatenate the profiles from three views to obtain a 192-dimensional embedding and perform clone detection with the same. Finally, the regular 64-dimensional multi-view embedding from apk2vec is also used for clone detection. From this experiment onwards, only WLK is considered for comparative evaluation, as it has the most consistent performance among the baselines.

Results & discussions. The results of this experiment are reported in Table 6.5. The following inferences are drawn from the table:

- At the outset, it is evident that individual views are capable of providing reasonable accuracies (i.e., 0.70+ ARI). This reveals that individual views possess capabilities to retain different, yet useful program semantics.

TABLE 6.6: Impact of semi-supervised embedding on malware detection efficacy

	apk2vec			WLK[121]		
Labels(%)	**P(%)**	**R(%)**	**F(%)**	**P(%)**	**R(%)**	**F(%)**
0	92.97	95.43	94.19	95.83	91.04	93.38
10	95.44	97.11	96.27	-	-	-
20	95.91	96.76	96.33	-	-	-
30	**96.59**	**97.28**	**96.93**	-	-	-

- Out of the individual views, as expected, API view get the best accuracy. This could be attributed to fact that this view extracts much larger number of high-quality features compared to the other two views. Owing to this well-known inference, many works in the past (e.g., [20, 24, 25, 43, 99]) have used them for a variety of tasks (incl. malware and clone detection). Also, source-sink view extract too few features to perform useful learning. In other words, it ends up underfitting the task. These observations are inline with the existing work on multi-view learning such as [18].
- In the case of `WLK`, API profiles gets a very high accuracy and when they are concatenated with other views, the accuracy is reduced. We believe this is due to the inherent linearity in this mode of combination i.e., views do not complement each other.
- Interestingly, in the case of `apk2vec`, concatenating profiles from individual views yields higher accuracy than using just one view. This reveals that using multiple views is indeed offering richer semantics and helps to improve accuracy. However, concatenation could only facilitate a linear combination of views and hence is yielding slightly lesser accuracy than `apk2vec`'s multi-view profiles. This illustrates the need for performing a non-linear combination of the semantic views.

6.5.3 Semi-supervised vs. Unsupervised Profiling

In this experiment, we intend to study the impact of using the (available) class labels of apps during profiling.

Dataset & experiments. Here, we use the same dataset which was used for online malware detection reported in Section 6.5.1.1. However, in order to study the impact of varying levels of supervision, we use labels for the following percentages of samples: 10%, 20%, and 30%. Profiles for apps are built with aforementioned

levels of supervision and for each setting an SVM classifier is trained and evaluated for malware detection (other settings such as train/test split are similar to Section 6.5.1.1).

Results & discussions. The results of this experiment are reported in Table 6.6, from which the following inferences are made:

- One could observe that leveraging semi-supervision during `apk2vec`'s embedding is indeed helpful in improving the accuracy of the downstream task. For instance, by using labels for merely 10% of samples help to improve the accuracy for malware detection by more than 2%.
- Clearly, using 30% labels yields better results than using just 10% and 20% labels. This illustrates the fact that the more the supervision is, the better the accuracy would be.
- In the case of `WLK` which uses handcrafted features, one could not use labels or other form of supervision to obtain graph embeddings. Hence, without any supervision, it performs reasonably well to obtain an f-measure of more than 93%. However, `apk2vec` with even just 10% supervision is able to outperform `WLK` significantly i,e., by nearly 3% f-measure.

6.5.4 Parameter Sensitivity

The `apk2vec` framework involves a number of hyperparameters such as embedding dimensions (δ), number of hash buckets (B) and number of hash functions (k). In this subsection, we examine how the different choices of δ affects `apk2vec`'s accuracy and efficiency, as it is the most influential hyperparameter.

Dataset & experiments. Here, the clone detection experiment reported in Section 6.5.1.2 is reused. The sensitivity results are fairly consistent on the remaining tasks reported in Section 6.5.1. Except for the parameter being tested, all other parameters assume default values. Embeddings' accuracy and efficiency are determined by ARI and pretraining durations (averaged over all epochs), respectively.

Results & discussions. These results are reported in Figure 6.3 from which the following inference are drawn.

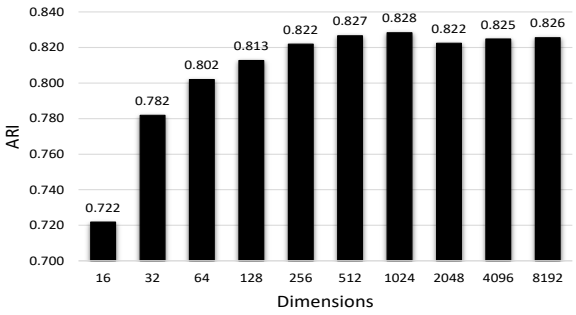


FIGURE 6.3: Sensitivity w.r.t embedding sizes

- Unsurprisingly, the ARI values increase with δ. This is understandable as larger embedding sizes offer better room for learning more features. However, the performance tends to saturate once the δ is around 500 or larger. This observation is consistent with other graph substructure embedding approaches [122, 125, 126].
- Also, the average pretraining time taken per epoch increases with δ. This is expected, since increasing δ would result in an exponential increase in skipgram computations. This is reflected in the exponential increase in pretraining time (especially, when $\delta > 500$). This analysis helps understanding the trade-off between `apk2vec`'s accuracy and efficiency for a given dataset and picking the optimal value for δ.

6.6 Conclusion

In this chapter, we presented `apk2vec`, semi-supervised multimodal RL technique to automatically build data-driven behavior profiles of Android apps. Through our large-scale experiments with more than 42,000 apps, we demonstrate that profiles generated by `apk2vec` are task agnostic and outperform existing approaches on several tasks such as malware detection, familial clustering, clone detection and app recommendation. Our semi-supervised multimodal embeddings also prove to provide significant advantages over their unsupervised and unimodal counterparts. All the code and data used within this work is made available at `https://sites.google.com/view/apk2vec/home`.

Chapter 7

Conclusion and Future Research

In this chapter, we summarise the research work that we have conducted in the thesis and discuss our future research directions.

7.1 Summary of completed work

Along with the rising popularity of Android, all of its stakeholders, such as developers and users, are facing increasing security threats that occur in different forms. In this thesis, we conducted a series of studies, based on program analysis and machine learning approaches to address these threats.

First, we observe that app cloning is a critical problem to the Android ecosystem as it affects multiple stakeholders. However, existing clone detection approaches lack resilient to obfuscation which are common in Android apps. To address this research gap, we propose a novel technique to detect app clones based on the analysis of UI information collected at runtime. Unlike the existing approaches, our approach is obfuscation resilient since the runtime behaviours are unaffected by the semantics preserving obfuscations. In addition, our approach leverage on the multiple entry point characteristic of Android apps and overcome the limitation of having to execute the entire app for runtime information collection.

Second, we propose a tool, named LibSift, to perform automated TPL detection as the presence of TPLs introduces privacy risks, security threats and hinders program analytics tasks. Existing works typically use a whitelist to match the package

names and exclude the TPLs from their analysis. However, package renaming obfuscation is commonly employed in Android apps. To this end, LibSift detects TPLs based on package dependencies that are resilient to most common obfuscations. Furthermore, LibSift does not depend on the assumption that the TPLs will be used in many apps. These allow LibSift to detection TPLs more effectively than existing approaches.

Third, we observe that the majority of the existing Android malware detection approaches are build upon ML algorithm. More specifically, they follow a batch learning model and erroneously assume that the population distribution is stationary. To this end, we performed a systematic study on a large dataset of over 80K apps to examine the limitations of state-of-the-art ML based malware detection approaches in presence of concept drift. We further propose modifications that can be applied to such approaches to help them overcome the observed limitations. Through several large-scale experiments, we demonstrated that our proposed modifications to the models significantly improve their performance.

Fourth, we observe that extensive feature engineering is required for each of the program analytic tasks that we have conducted. The set of features thus obtained are specific to the respective analytics task and may not be transferable to other tasks. To this end, we propose a static analysis based semi-supervised multi-view RL framework named `apk2vec`, to build holistic profiles of Android apps that are capable of catering to multiple downstream analytics tasks. The results of our extensive experiment on more than 42K apps demonstrate that our app profiles could outperform state-of-the-art solutions in four program analytic tasks, such as, malware detection, familial classification, app clone detection and app recommendation.

7.2 Future work

In this thesis, we have developed techniques to address different security challenges in the Android ecosystem and also proposed an approach to build holistic Android app profile that can catering a range of downstream analytic tasks. However, due to the magnitude of the challenges,there are still many issues that are not covered in this thesis. In future, we are going to conduct the following works:

- **Graph attention model for Android malware analysis.** In our work on `apk2vec` (see Chapter 6), the entire graph representation is used to perform malware detection. However, in reality, only a small portion of the graph represents the malicious code fragment. Inspired by the recent success of RNN with attention on vision and natural language processing tasks, we envision to employ a RNN model with attention mechanism to build a app representation for malware analysis. Using attention to focus on the malicious part of the ICFG allows us to on the informative part of the graph and improve detection accuracy while also enabling us to identify the portion of the graph that is malicious.

- **Enhance app profile with metadata.** Apart from bytecode, there are also other sources of information that describe the apps' behaviour. For instance, the app description on the app market and image in the resource files, both provide clues on the app's functionality. To further enhance the capabilities of our app profiles, we plan to extend `apk2vec` with CNN for image and RNN for text processing. Having such app profiles can improve the performance on the downstream analytic tasks and open up to more potential applications such as app categorisation.

- **Cross-platform mobile app representation learning.** In comparison with other smartphone platforms, the apps from Android platform are predominately easier to be reverse engineered and analysed. However, libraries or code fragment typically have counterparts in other platforms. To this end, we intend to study the association between Android and iOS apps and use this knowledge for performing cross-platform app representation learning. This can be achieved by incorporating machine translation and transfer learning techniques to our app profiling approach.

Author's Publications

Journal Articles

- Arnatovich, Yauhen Leanidavich, Lipo Wang, Ngoc Minh Ngo, and **Charlie Soh**. "A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation." IEEE Access 6 (2018): 12382-12394.
- Arnatovich, Yauhen Leanidavich, Lipo Wang, Ngoc Minh Ngo, and **Charlie Soh**. "Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation." Software: Practice and Experience, 2018.

Conference Proceedings

- **Soh, Charlie**, Annamalai Narayanan, Lihui Chen, Lipo Wang and Yang Liu. "apk2vec: A Multi-view Deep Learning Framework for Profiling Android Applications." Data Mining (ICDM), 2018 IEEE International Conference on. IEEE, 2018. (Accepted)
- **Soh, Charlie**, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. "LibSift: Automated Detection of Third-Party Libraries in Android Applications." In Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific, pp. 41-48. IEEE, 2016.
- Arnatovich, Yauhen Leanidavich, Minh Ngoc Ngo, Tan Hee Beng Kuan, and **Charlie Soh**. "Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising." In Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific, pp. 193-200. IEEE, 2016.

- **Soh, Charlie**, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. "Detecting clones in android applications through analyzing user interfaces." In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, pp. 163-173. IEEE Press, 2015.

Bibliography

[1] An overview of the android architecture, . URL `http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture,`. ix, 12

[2] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016. ix, 6, 45, 46, 55, 56, 59

[3] Li Li, Jacques Klein, Yves Le Traon, et al. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414. IEEE, 2016. ix, 6, 45, 46, 55, 56, 59, 60

[4] 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 (DIMVA)*, pages 252–276. Springer, 2017. xii, 69, 75, 84

[5] Idc:worldwide smartphone os market share. URL `https://www.idc.com/promo/smartphone-market-share/os`. 1

[6] Appbrain: Google play stats, . URL `http://www.appbrain.com/stats`. 1

[7] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer, 2012. 1

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth.

Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014. 20

[9] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12:110, 2012. 1

[10] Kangjie Lu, Zhichun Li, Vasileios P Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*, 2015. 1

[11] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 252–261. IEEE, 2016. 1, 61, 62, 69

[12] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[13] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

[14] Amit Deo, Santanu Kumar Dash, Guillermo Suarez-Tangil, Volodya Vovk, and Lorenzo Cavallaro. Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, pages 71–82. ACM, 2016. 61, 69

[15] Guillermo Suarez-Tangil, Juan E Tapiador, and Pedro Peris-Lopez. Stegomalware: Playing hide and seek with malicious components in smartphone apps. In *International Conference on Information Security and Cryptology*, pages 496–515. Springer, 2014. 61

[16] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017. 1, 61, 68, 69, 89

[17] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175, 2017. 3

[18] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. A multi-view context-aware approach to android malware detection and malicious code localization. *Empirical Software Engineering*, pages 1–53, 2017. 92, 114

[19] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014. 20, 22, 36, 41, 44, 58, 92, 109, 111

[20] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014. 20, 62, 92, 114

[21] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013. 20, 92

[22] Ke Tian, Danfeng Daphne Yao, Barbara G Ryder, G Tan, and Guojun Peng. Detection of repackaged android malware with code-heterogeneity features. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[23] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 14–24. IEEE Press, 2017.

[24] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 12(8):1772–1785, 2017. 114

[25] 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. *IEEE Transactions on Information Forensics and Security*, 2018. 92, 114

[26] 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 Security Symposium*, volume 15, 2015. 3, 42, 92

[27] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 13, 18

[28] ¡activity¿. URL `https://developer.android.com/guide/topics/manifest/activity-element`. 14, 26

[29] Android and security. URL `http://googlemobile.blogspot.sg/2012/02/android-and-security.html`. 16

[30] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012. 4, 16, 22, 40, 44, 58

[31] dex2jar: Tools to work with android .dex and java .class files. URL `https://github.com/pxb1988/dex2jar`. 17, 21

[32] Soot: A framework for analyzing and transforming java and android applications. URL `https://sable.github.io/soot/`.

[33] smali: An assembler/disassembler for android's dex format. URL `https://github.com/JesusFreke/smali`. 17, 48

[34] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014. 17, 42

[35] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, page 10. Citeseer, 2012. 19, 44

[36] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[37] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016. 19, 59

[38] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251. ACM, 2014. 19, 41, 43

[39] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82. ACM, 2015. 4, 19, 20, 41, 43, 58, 59

[40] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*. Citeseer, 2015. 19, 44

[41] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014. 19, 44

[42] 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, pages 185–196. ACM, 2013. 20, 41, 45, 46, 58

[43] 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*, 2014. 20, 62, 63, 64, 66, 92, 109, 110, 111, 114

[44] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 303–313. IEEE Press, 2015.

[45] 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. 20

[46] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011. 20

[47] Apktool: A tool for reverse engineering android apk files. URL `http://ibotpeaches.github.io/Apktool/`. 21, 48, 52

[48] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, page 6. ACM, 2012. 21

[49] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444. ACM, 2013. 21

[50] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012. 21, 22, 61, 75, 89

[51] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*, pages 37–54. Springer, 2012. 4, 22, 40, 44, 45, 58

[52] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A framework for evaluating mobile app repackaging detection algorithms. In *International Conference on Trust and Trustworthy Computing*, pages 169–186. Springer, 2013. 22

[53] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2012. 22, 40

[54] Activities. URL `https://developer.android.com/guide/components/activities`. 24

[55] Test apps on android. URL `https://developer.android.com/training/testing/#UIAutomator`. 24

[56] Android debug bridge. URL `https://developer.android.com/studio/command-line/adb`. 26

[57] App resources overview. URL `https://developer.android.com/guide/topics/resources/providing-resources#Accessing`. 28

[58] Lsh algorithm and implementation (e2lsh). URL `http://www.mit.edu/~andoni/LSH/`. 29

[59] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004. 29

[60] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955. 30

[61] Free vs. paid android apps, . URL `http://www.appbrain.com/stats/free-and-paid-android-applications`. 32

[62] Anruan. URL `http://www.anruan.com`. 33

[63] Appsapk, . URL `http://www.appsapk.com`. 33

[64] Pandaapp. URL `http://download.pandaapp.com`. 33

[65] Virustotal, . URL `https://www.virustotal.com`. 38, 65, 67, 74

[66] Androguard: Reverse engineering, malware and goodware analysis of android applications, . URL `https://github.com/androguard/androguard/`. 40

[67] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 638–648. ACM, 2017. 4, 41, 58

[68] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: fast detection of repackaged applications. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 130–145. Springer, 2014. 41

[69] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49(6):259–269, 2014. 44

[70] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014. 44

[71] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015. 44, 61, 62, 66

[72] 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. 4, 44, 58

[73] Wenhui Hu, Damien Octeau, Patrick Drew McDaniel, and Peng Liu. Duet: library integrity verification for android applications. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 141–152. ACM, 2014. 44

[74] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012. 44

[75] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following devils footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *IEEE Symposium on Security and Privacy*. in press, 2016. 4, 44, 58, 92

[76] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *European Symposium on Research in Computer Security*, pages 182–199. Springer, 2013. 58

[77] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014. 58

[78] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*, page 3. ACM, 2014. 59, 62

[79] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346. IEEE, 2017. 59

[80] Symantec: Internet security threat report. URL `https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf`. 61

[81] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012. 61, 62

[82] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012. 62

[83] Wen-Chieh Wu and Shih-Hao Hung. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, pages 247–252. ACM, 2014. 61, 62

[84] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41 (4):1104–1117, 2014. 62

[85] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015. 62

[86] Kevin Allix, Tegawendé François D Assise Bissyande, Quentin Jerome, Jacques Klein, Yves Le Traon, et al. Empirical assessment of machine learning-based malware detectors for android: Measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering*, pages 1–29, 2014. 62, 63, 64, 65, 66, 67

[87] Brandon Amos, Hamilton Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless communications and mobile computing conference (iwcmc), 2013 9th international*, pages 1666–1671. IEEE, 2013. 62

[88] Annamalai Narayanan, Liu Yang, Lihui Chen, and Liu Jinliang. Adaptive and scalable android malware detection through online learning. In *Neural Networks (IJCNN), 2016 International Joint Conference on*, pages 2484–2491. IEEE, 2016. 62

[89] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101, 1996. 63

[90] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2), 2004. 63

[91] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015. 64, 65, 89

[92] Kevin Allix, Tegawendé François D Assise Bissyande, Jacques Klein, and Yves Le Traon. Machine learning-based malware detection for android applications: History matters! Technical report, University of Luxembourg, SnT, 2014. 66, 71

[93] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007. 69

[94] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009. 70

[95] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D Joseph, and JD Tygar. Approaches to adversarial drift. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 99–110. ACM, 2013. 71

[96] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time url spam filtering service. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 447–462. IEEE, 2011. 71

[97] J Gama, P Medas, G Castillo, and PP Rodrigues. Learning with drift detection, 2004. 71

[98] Anshuman Singh, Andrew Walenstein, and Arun Lakhotia. Tracking concept drift in malware families. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 81–92. ACM, 2012. 72

[99] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175, 2017. 72, 92, 101, 110, 114

[100] Rajasekar Venkatesan and Meng Joo Er. A novel progressive learning technique for multi-class classification. *Neurocomputing*, 207:310–321, 2016. 73, 84

[101] Google play. URL `https://play.google.com/,`. 74, 109

[102] Anzhi. URL `http://www.anzhi.com/,`. 74

[103] Appchina, . URL `http://www.appchina.com/,`. 74

[104] Slideme. URL `http://slideme.org/,`. 74

[105] Hiapk. URL `https://play.google.com/,`. 74

[106] Fdroid. URL `https://f-droid.org/en/,`. 74

[107] Angeeks. URL `http://bbs.angeeks.com/portal.php,`. 74

[108] 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*, 21(1):183–211, 2016. 89

[109] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*, pages 3–17. IEEE, 2014. 89

[110] Wenchao Li, Hassen Saidi, Huascar Sanchez, Martin Schäf, and Pascal Schweitzer. Detecting similar programs via the weisfeiler-leman graph kernel. In *International Conference on Software Reuse*, pages 315–330. Springer, 2016. 92

[111] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013. 92, 101

[112] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 93, 95

[113] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017. 93, 97, 106, 108, 111, 112, 113

[114] Dan Tito Svenstrup, Jonas Hansen, and Ole Winther. Hash embeddings for efficient word representations. In *Advances in Neural Information Processing Systems*, pages 4928–4936, 2017. 93, 97

[115] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014. 96

[116] Bijaya Adhikari, Yao Zhang, Naren Ramakrishnan, and B Aditya Prakash. Distributed representations of subgraphs. In *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*, pages 111–117. IEEE, 2017. 98, 108, 111, 112, 113

[117] Dang Nguyen, Wei Luo, Tu Dinh Nguyen, Svetha Venkatesh, and Dinh Phung. Learning graph representation via frequent subgraphs. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 306–314. SIAM, 2018. 98, 108, 109, 110, 112

[118] Sergey Ivanov and Evgeny Burnaev. Anonymous walk embeddings. *arXiv preprint arXiv:1805.11921*, 2018. 98

[119] Shirui Pan, Jia Wu, Xingquan Zhu, Chengqi Zhang, and Yang Wang. Tri-party deep network representation. In *Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, 2016. 98

[120] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012. 101

[121] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011. 104, 108, 111, 112, 113, 114

[122] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM, 2015. 104, 109, 116

[123] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and SVN Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10(Nov):2615–2637, 2009. 105

[124] Virus share malware repository, . URL `https://virusshare.com,`. 109

[125] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016. 112, 116

[126] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014. 116