Improving Android App Quality Using Big Code Analysis-based Techniques

Yanjie Zhao

Faculty of Information Technology

Monash University

A thesis submitted for the degree of

Doctor of Philosophy

February 2023

Copyright

Abstract

There has been a significant rise in smartphone users between 2017 and 2022, with a 49.89% growth in the number of individuals who possess a smartphone. With the mobile industry's development, the competition has become increasingly intense, whether between app markets or apps of the same category. Ensuring the high quality of apps in order for them to remain competitive has become a top priority for app markets and app developers. However, this is not easy as app markets are constantly challenged by malware and app developers are often under pressure to fix vulnerabilities, solve compatibility issues, and learn new development methodologies, libraries, and the latest technologies to keep up with the rapidly evolving mobile landscape.

Just like big data that brings vast amounts of data enabling big data applications, large repositories of Android app programs (e.g., open-source code, Dalvik bytecode, Java bytecode) allow a new class of applications that leverage these repositories of "big code". This thesis seeks to utilize big code analysis-based techniques to improve the quality of Android apps.

For app markets, the importance of malware detection can never be overstated since malicious apps often lead to a poor user experience. We first conduct a large-scale analysis based on the big code to assess the evaluation fairness of current malware detection approaches, where duplication was found in several widely popular Android datasets. This analysis calls for researchers to pay attention to the reliability of the datasets to ensure a fair evaluation of malware detection techniques.

For app developers, we aim to improve their developed apps' quality by reducing the stress of the development process and aiding bug fixing through big code analysis-based techniques. Providing readily reusable functions, APIs have become one of the most critical components in modern software development. Learning API usages from big code has gradually become a popular topic. We propose a novel tool for recommending code implementations for Android app development. Moreover, automated bug detection and fixing is another topic we are interested in. For example, the heavy fragmentation of the Android ecosystem has led to severe compatibility issues with apps, including those that crash at runtime or cannot be installed on specific devices but work well on other devices. To address this, we propose a generic approach that aims to fix different types of compatibility issues in released Android apps, reducing the burden on app developers, and improving the user experience.

Overall, leveraging big code analysis-based techniques is a promising approach for improving the quality of Android apps, benefiting both app markets and developers. Further research in this area can pave the way for more effective solutions to the challenges faced by the mobile industry.

Declaration of Authorship

I, Yanjie Zhao, declare that this thesis titled "Improving Android App Quality Using Big Code Analysis-based Techniques" and the work presented in it are my own. I confirm that:

- This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution.
- This work was done wholly or mainly while in candidature for a research degree at Monash University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have clearly shown what was done by others and what I have contributed myself.

Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 3 original papers published in peer reviewed journals/conferences and 0 submitted publications. The core theme of the thesis is "Improving Android App Quality Using Big Code Analysis-based Techniques." The ideas, development and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the Department of Software Systems and Cybersecurity, under the supervision of Prof. John Grundy and Dr. Li Li.

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research.

In the case of Chapter 3-5, my contribution to the work involved the following:

Thesis Chapter	**Publication Title**	**Status** *(published, in press, accepted or returned for revision, submitted)*	**Nature and % of student contribution**	**Co-author name(s) Nature and % of Co-author's contribution***	**Co-author(s), Monash student Y/N***
Chapter 3	On the Impact of Sample Duplication in Machine-Learning-Based Android Malware Detection	Published	60%. Concept, All of the experiment, writing the manuscript and revisions for the paper.	1) Li Li, Concept, input into manuscript and discuss with key challenges, 10% 2) Haoyu Wang, Concept and discuss with key challenges, 8% 3) HAIPENG CAI, discuss with key challenges, 2% 4) Tegawendé F. Bissyandé, input into manuscript and discuss with key challenges, 10% 5) Jacques Klein, discuss with key challenges, 2% 6) John Grundy, discuss with key challenges, 8%	N

Chapter 4	APIMatchmaker: Matching the Right APIs for Supporting the Development of Android Apps	Published	60%. Concept, All of the experiment, writing the manuscript and revisions for the paper.	1) Li Li, Concept, input into manuscript and discuss with key challenges, 15% 2) Haoyu Wang, discuss with key challenges, 10% 3) Qiang He, discuss with key challenges, 5% 4) John Grundy, discuss with key challenges, 10%	N
Chapter 5	Towards Automatically Repairing Compatibility Issues in Published Android Apps	Published	60%. Concept, All of the experiment, writing the manuscript and revisions for the paper.	1) Li Li, Concept, input into manuscript and discuss with key challenges, 20% 2) Kui Liu, input into manuscript and discuss with key challenges, 10% 3) John Grundy, discuss with key challenges, 10%	N

I have renumbered sections of submitted or published papers in order to generate a consistent presentation within the thesis.

Student name: Yanjie Zhao **Signature:**  **Date:** 27-Feb-2023

The undersigned hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

Main Supervisor name: John Grundy **Signature:**  **Date:** 28-Feb-2023

Publications

Publications included in this thesis

1. **Yanjie Zhao**, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F. Bissyandé, Jacques Klein and John Grundy, On the Impact of Sample Duplication in Machine Learning based Android Malware Detection, ACM Transactions on Software Engineering and Methodology (TOSEM), 2021. (Chapter 3)

2. **Yanjie Zhao**, Li Li, Haoyu Wang, Qiang He and John Grundy, APIMatchmaker: Matching the Right APIs for Supporting the Development of Android Apps, IEEE Transactions on Software Engineering (TSE), 2021 (Chapter 4)

3. **Yanjie Zhao**, Li Li, Kui Liu and John Grundy, Towards Automatically Repairing Compatibility Issues in Published Android Apps, In The 44th International Conference on Software Engineering (ICSE), 2022 (Chapter 5)

4. **Yanjie Zhao**, Li Li, Xiaoyu Sun, Pei Liu and John Grundy, Icon2Code: Recommending Code Implementations for Android GUI Components, Information and Software Technology (IST), 2021 (Chapter 6)

5. **Yanjie Zhao**, Li Li, Xiaoyu Sun, Pei Liu and John Grundy, Code Implementation Recommendation for Android GUI Components, In The 44th International Conference on Software Engineering (ICSE), Demonstrations Track, 2022 (Chapter 6)

Other publications during candidature

1. **Yanjie Zhao**, Tianming Liu, Haoyu Wang, Yepang Liu, John Grundy and Li Li, Are Mobile Advertisements in Compliance with App's Age Group? In The ACM Web Conference (WWW), Co-first author, 2023

2. Pei Liu, **Yanjie Zhao**, Li Li, Haipeng Cai, Mattia Fazzini and John Grundy, Automatically Detecting API-induced Compatibility Issues in Android Apps: A Comparative Analysis (Replicability Study), In The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Co-first author, 2022

3. Xiaoyu Sun, Xiao Chen, **Yanjie Zhao**, Pei Liu, John Grundy and Li Li, Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues, In the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2022

4. Pei Liu, Li Li, **Yanjie Zhao**, Xiaoyu Sun and John Grundy, Androzooopen: Collecting large-scale open source android apps for the research community, In Proceedings of the 17th International Conference on Mining Software Repositories (MSR), 2020

5. **Yanjie Zhao**, Haoyu Wang, Lei Ma, Yuxin Liu, Li Li and John Grundy, Knowledge graphing git repositories: A preliminary study, In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), ERA Track, 2019

Acknowledgements

First, my supervisors, Dr. Li Li and Prof. John Grundy, deserve my deepest thanks and admiration for their patience and attentive advice during my Ph.D. studies. I was given this wonderful opportunity and was encouraged to come to Monash University to pursue my doctoral degree by my supervisors. Throughout my Ph.D. journey, they provided me with exceptional assistance and supported me with great kindness. I would not be able to address the software engineering challenges without their unending assistance. They've shown me how to research, write a technical report, conduct engaging presentations, and complete idea generation and implementation. My weekly/bi-weekly meetings with them provided a lot of motivation and inspiration for my Ph.D. research.

Secondly, I want to convey my appreciation to all of my co-authors, especially Prof. Tegawendé F. Bissyandé, Prof. Jacques Klein, Prof. Haoyu Wang, and Dr. Kui Liu. They have contributed insightful conversations and recommendations for our joint endeavors. Their in-depth critiques have strengthened our work and greatly elevated its quality.

Furthermore, I would like to express my gratitude to the chair and panel members of my milestones, A/Prof. Rashina Hoda, Dr. Chunyang Chen, and Dr. Chakkrit Tantithamthavorn, as they provided me with insightful comments on the three milestones during my Ph.D. studies. Additionally, I would like to thank the whole academic staff and administrative staff of the Faculty of Information Technology.

I would also like to express my heartfelt gratitude to my fellow lab mates, especially Xiaoyu Sun, Pei Liu, and Tianming Liu, who have collaborated closely with me throughout

the entire research process. Their invaluable insights, constructive feedback, and unwavering support have greatly contributed to the successful completion of this study. I am fortunate to have had the opportunity to work alongside such talented and dedicated individuals, and I look forward to our continued collaboration in future endeavors.

Finally, I want to thank my parents for their unwavering support and encouragement, and for assuring me that they would be there for me no matter my difficulties.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Key Research Questions

Motivation. Mobile apps are becoming an indispensable part of our daily lives, enhancing our productivity, entertainment, and communication. With over 2.8 billion active users globally, Android has emerged as the dominant mobile operating system [37], providing a plethora of features and offering immense flexibility for developers to create innovative and useful apps. Nevertheless, the increasing popularity of Android has also brought with it a host of challenges in terms of app quality. Android apps are susceptible to various issues, such as malware attacks, compatibility issues, suboptimal application programming interface (API) usage, and unsatisfactory user interfaces. These problems negatively impact the user experience and result in substantial revenue losses for app markets, developers, and the broader Android app ecosystem. Thus, there is an urgent need to enhance the quality of Android apps.

For *app markets*, malware detection is of critical importance, as the consequences of allowing malicious apps to infiltrate can be dire. Malware not only leads to a poor user experience but can also result in severe security breaches, data theft, and financial loss. Furthermore, the reputation of app markets can be irreparably damaged by the presence of malicious apps, leading to a loss of user trust and a decline in user engagement. Thus, it is vital for app markets to prioritize the detection and removal of malware from their

platforms to provide their users with a safe and secure app ecosystem.

For *app developers*, the task of developing high-quality Android apps can be incredibly challenging and complex. It involves a variety of tasks, including designing user interfaces, writing efficient code, testing for bugs and errors, and deploying software that works seamlessly across a wide range of devices and operating systems. To achieve success in this fiercely competitive marketplace, developers need to manage a short app release cycle, stay abreast of the latest technologies and best practices, and continually improve their skills to meet the evolving needs of their users.

Just like big data that brings vast amounts of data enabling big data applications, large repositories of programs (e.g., open-source code, Dalvik bytecode, Java bytecode) allow a new class of applications that leverage these repositories of "big code". One promising approach to improving Android app quality is through big code analysis-based techniques, which offer new opportunities to identify and address the underlying issues that contribute to poor app quality. Specifically, big code analysis-based techniques can help improve the quality of Android apps by analyzing large volumes of code and identifying patterns, trends, and potential issues that may affect the app's performance, security, or user experience.

The primary focus of this thesis is to explore the application of big code analysis-based techniques for improving the quality of Android apps. Specifically, we focus on the following three key issues and observations:

Malware is a significant concern for Android users, as it has the potential to steal personal information, install unwanted software, or even take control of a device. With the increasing prevalence of mobile devices, the need for reliable and effective malware detection approaches has never been greater, and app markets must remain vigilant in their efforts to protect their users from potential harm. Machine learning (ML) techniques are often used to detect malware at scale in the Android realm, but the effectiveness of these

approaches is heavily reliant on the quality of the datasets used for training and validation, as well as the evaluation procedures employed. Moreover, in the context of malware detection, big code analysis can be used to extract relevant features that can be used to improve the effectiveness of machine learning-based approaches. Unfortunately, many datasets used in the app analysis community may contain a large portion of duplicated samples, which can bias recorded experimental results and insights. Therefore, ensuring the reliability of big code is the first step in conducting ML-based malware detection.

Poor API usage can significantly impact the quality and performance of Android apps. APIs have become one of the most important components in modern software development, providing readily reusable functions that support a wide range of functionalities such as navigating maps, supporting security, using device features, and processing images and voice, among others. While the availability of a large number of existing libraries provides convenience for experienced developers to implement software quickly, it also introduces significant burdens for developers, who must constantly spend a significant amount of time learning the usage of each new API in detail to correctly deploy it. Unfortunately, some developers may not always use APIs effectively, leading to inefficient code and performance issues. The application of big code analysis in this domain can surface aggregated patterns of API usage across projects, offering unprecedented insights into what constitutes effective or poor API usage. This is invaluable for developers aiming for optimal performance and code quality. With the help of big code analysis-based techniques, developers can gain insights into how to use APIs more effectively, identify common mistakes and best practices, and optimize their code to improve the overall quality of their Android apps. By addressing poor API usage, developers can create better-performing apps that provide a superior user experience and meet the evolving needs of their users.

Compatibility issues are a common problem that Android app developers encounter. The fragmentation of the Android platform requires developers to ensure that their apps

function correctly across a wide range of devices and operating systems. To address this challenge, various approaches have been proposed to automatically detect and fix compatibility issues. However, these all come with various limitations on fixing the compatibility issues, e.g., can only fix one specific type of issues, cannot deal with multi-invocation issues in a single line and issues in released apps. Big code analysis provides a unique advantage here as well; by analyzing a large corpus of apps, it allows for the identification of recurring compatibility issues across different versions and devices, thereby enabling the development of more comprehensive compatibility solutions. By leveraging big code analysis-based techniques, developers can gain insights into the common compatibility issues that exist across a range of devices and operating systems. This knowledge can be used to inform the development of more effective techniques for detecting and fixing compatibility issues automatically. Improving compatibility is an essential step toward creating high-quality Android apps that provide a seamless and consistent experience for users across a diverse range of devices and operating systems.

Research Questions. Given the aforementioned issues and observations about the Android ecosystem, this thesis raises three key research questions aimed at improving the overall quality, reliability, and security of Android apps. These research questions are as follows:

- **Research Question 1:** Can we ensure the reliability of big code used for machine learning-based malware detection approaches and their evaluation procedures? Are they susceptible to the influence of big code quality, such as sample duplication?
- **Research Question 2:** How can big code analysis-based techniques be used to provide API usage recommendations for improving app development?
- **Research Question 3:** How can big code analysis-based techniques be used to help in repairing compatibility issues in Android apps?

Overall, the research objectives of this thesis aim to provide insights into how to ensure the reliability of big code, how big code analysis-based techniques can be used to improve Android app quality, and how developers can properly apply these techniques to specific areas of app development to enhance the user experience and ensure the reliability and security of their apps.

1.2 Thesis Scope

In this thesis, we aim to provide valuable insights, guidance, and assistance for the Android community, which will foster the development of higher-quality and more secure Android apps. In particular, to address the aforementioned research questions, we take three steps, each focused on a specific area of investigation:

- **Sample duplication**: big code reliability analysis to benefit malware detection;
- **APIMatchmaker**: recommending APIs for better Android development; and
- **RepairDroid**: supporting automatically repairing compatibility issues in apps.

Sample duplication (Chapter 3). To ensure unbiased studies are carried out in this domain, we begin by performing a reliability analysis on big code-based analysis systems. In order to explore the connection between Android apps at the code level for subsequent research on API recommendation and bug fixing (using big code analysis techniques to learn knowledge from similar apps), we are first concerned with the phenomenon of *sample duplication*. Following our preliminary analysis, we identified a prevalence of duplication within popular Android datasets. Machine learning (ML) techniques are widely used for malware detection in the Android realm [13, 110]. However, many commonly used datasets may contain duplicated samples, which can bias experimental results and impact the quality of ML-based models. Despite the critical role of datasets in the training and validation

of ML models, little attention has been paid to their intrinsic quality beyond issues such as class imbalance and temporal alignment [8, 7, 130]. Allamanis [6] has highlighted the issue of code sample duplication, where a given sample is repeated multiple times in a dataset. Our work aims to better understand the impact of sample duplication on malware detector performance with big code analysis (both static analysis and machine learning-based analysis), given the potential presence of duplicated features in Android samples extracted from code snippets and metadata such as permissions and resource files.

- *This work has led to a research paper –* ***On the Impact of Sample Duplication in Machine-Learning-Based Android Malware Detection****, which has been published in the ACM Transactions on Software Engineering and Methodology (TOSEM) in 2021.*

APIMatchmaker (Chapter 4). After conducting big code reliability analysis, we then want to explore how big code analysis-based techniques can contribute to improving the quality of Android apps. Since duplication in big code is prevalent, we can easily learn useful knowledge from other similar apps to help Android development. APIs are essential for modern software development as they offer pre-built, reusable functions. However, with the vast number of libraries available, developers may face challenges in mastering each new API in order to apply them correctly. To tackle this issue, we create a prototype API recommendation system, i.e., **APIMatchmaker**, to support Android app development. Through a static analysis based on big code, we can extract their API call usages from a vast number of apps to build such a code recommendation system. This system can learn and suggest optimal Android API usage patterns for each editing method. The goal is to relieve the burden of learning API specifics and, ultimately, to enhance the overall quality of Android apps.

- *This work has led to a research paper –* ***APIMatchmaker: Matching the Right APIs***

***for Supporting the Development of Android Apps**, which has been published in IEEE Transactions on Software Engineering (TSE) in 2021.*

RepairDroid (Chapter 5). Leveraging big code analysis-based techniques to help automated bug and issue repair is another topic we are interested in. One specific area of focus is the fragmentation of the Android ecosystem, which can result in compatibility issues that cause apps to crash or fail to install on certain devices. We propose a prototype tool called **RepairDroid**, which allows users to create fix templates for compatibility issues using a generic app patch description language. These templates are then leveraged to automatically fix the corresponding issue at the bytecode level, right before the app is installed. The tool can create templates for OS-induced, device-specific, and inter-callback compatibility issues detected by three state-of-the-art approaches. In order to make this framework practical and effective, it is necessary to extract and learn from real-world Android apps using big code analysis-based techniques. This prerequisite step is critical in enabling the framework to learn from actual code and identify patterns and relationships that can be used to identify and fix bugs.

- *This work has led to a research paper – **Towards automatically repairing compatibility issues in published Android apps**, which has been published at the 44th International Conference on Software Engineering (ICSE) in 2022.*

1.3 Thesis contribution

This section summarizes the main contributions of the thesis. This thesis builds on top of materials from our past papers about **Sample duplication**, **APIMatchmaker**, and **RepairDroid**.

Chapter 3 – **Sample duplication** – describes our extensive experiments to measure the performance gap that occurs when datasets are de-duplicated. In this work, we make the

following key contributions:

1. We empirically investigated the impact of sample duplication on machine learning-based Android malware detection approaches. We started by recognizing common sample duplication types in well-known and used Android malware datasets through big code analysis-based techniques.

2. We then took into account these sample duplication types to train distinctive machine-learning models to classify Android malware. We conducted our experiments on three common malware datasets. Our experimental results show that sample duplication does indeed impact the performance of machine learning-based malware detection approaches.

3. An in-depth exploration further revealed that this finding applied to not only in-the-lab experiments (i.e., 10-fold cross-validation) but also in-the-wild analyses (i.e., trained on one dataset and then tested on another). This finding also applies to experiments that were conducted using different machine learning algorithms, including both supervised and unsupervised learning approaches.

Chapter 4 introduces the novel tool **APIMatchmaker**. This aims to recommend better API usages to developers by learning directly from similar real-world Android apps. In this work, we make the following key contributions:

1. We introduce to the community a new multidimensional context-aware collaborative filtering approach to better analyze the big code and locate the most similar apps to support the recommendation.

2. We design and implement a prototype tool called **APIMatchmaker**, which takes as input a method under editing and outputs a list of APIs (and their usage samples)

meeting the constraints of the SDK versions that could be leveraged to complete the implementation of the method.

3. We evaluate our approach on 12,000 real-world Android apps under different experimental settings. Experimental results show that our big code analysis-based approach is promising in recommending API usage to Android app developers.

Chapter 5 introduces the novel tool **RepairDroid**, which provides a generic app patch description language for users to create fix templates for compatibility issues. The created templates will then be leveraged by **RepairDroid** to automatically fix the corresponding issue at the bytecode level. In this work, we make the following key contributions:

1. We have designed a novel app patch description language and demonstrated that it is generic enough to be used to create fix templates for various compatibility issues. The genericity is achieved by allowing users to directly leverage the simple but well-defined Jimple grammar (i.e., a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java/Android bytecode) to describe the patches.

2. We have designed and implemented a prototype tool **RepairDroid**, which follows given fix templates to repair published real-world Android apps automatically.

3. We have evaluated our approach against 1,000 real-world Android apps. Experimental results show that our approach, which is based on the knowledge learned through big code analysis-based techniques, is effective in repairing Android apps, outperforms the state-of-the-art, and achieves an 85.34% successful repairing rate.

In summary, this Ph.D. thesis conducted three studies targeting our three research objectives, respectively. With the help of our research, (i) researchers can get insights about how

to investigate the impact of dataset bias, e.g., **sample duplication**, on machine learning-based Android malware detection with big code analysis-based techniques; (ii) developers can use our tool **APIMatchmaker** to obtain API usage recommendations by learning directly from similar real-world Android apps that are selected by big code analysis; (iii) developers and app market managers can automatically repair compatibility issues in the published Android apps by using **RepairDroid**, which directly applies the templates created based on the knowledge learned from big code to the Android apps.

1.4 Structure of the thesis

In this thesis, Chapter 1 provides an introduction and motivation for the research conducted. Chapter 2 presents a literature review, including a discussion on Android malware detection & bias in machine learning, recommendations for Android development, and compatibility issues in Android apps. In Chapter 3, extensive experiments are conducted to measure the impact of dataset de-duplication on the performance of machine learning-based Android malware detection. Chapter 4 introduces **APIMatchmaker**, an approach for recommending API usage patterns by learning from similar real-world Android apps. Chapter 5 presents **RepairDroid**, a tool that enables automatic repair of compatibility issues in published Android apps through a generic app patch description language for users to create fix templates. In Chapter 6, we offer a conclusion to the thesis by summarizing the key contributions and limitations and presenting directions for future research.

Chapter 2

Literature Review

In this chapter, we elucidate the essential background required to comprehend the objectives, key challenges, and technical intricacies of the three research studies showcased in this thesis. A systematic literature review was conducted, which we categorized into three principal areas: Android malware detection & bias in machine learning, recommendations for Android development, and compatibility issues in Android apps. The aim of this review is to offer readers a comprehensive and organized overview of current research in the field, as well as suggest potential avenues for future research.

2.1 Malware Detection & Bias in Machine Learning

Prior research in the software engineering and security community has delved deeply into machine learning-based techniques for detecting Android malware. The significance of datasets in training and evaluation has been recognized as a crucial aspect of this research. In the following, we provide an overview of some noteworthy previous studies.

Machine learning-based Android malware detection. Machine learning has been extensively leveraged by practitioners and researchers to detect Android malware [128, 175]. One of the most common algorithms leveraged by researchers for achieving this purpose is RandomForest, which has been reported by researchers as one of the best algorithms for

conducting binary classification. As an example, Alam et al. [5] have empirically demonstrated that RandomForest is optimal by comparing its accuracy with BayesNet, Logistic Regression, DT, etc. Later on, Allix et al. [7] have also empirically confirmed this. In their experiment, they experimentally show that RF achieves the best performance compared with C4.5, RIPPER, and SVM. A similar result has also been backed up by Li et al. [82] as well.

SVM has been a very common machine learning algorithm for training to predict Android malware. For example, Naser et al. [129] built a malware detector based on the features statically extracted from Android APKs. One of the most famous works that leverage SVM to predict Android malware is the one presented by Arp et al. [13]. They proposed the Drebin approach, for which they extract machine learning features from Android APKs (or DEX files) into eight feature sets.

Most of the aforementioned approaches extract features statically from Android DEX files, which contain the core app code of the apps. Indeed, as an example, Jerome et al. [68] have proposed an ML-based malware detection approach based on opcode sequences in 2014. Similar to their work, Canfora et al. [27] and McLaughlin et al. [111] also respectively present machine learning-based malware detection approaches based on features statically extracted from the raw Dalvik bytecode (i.e., opcode sequences). Since similar opcode sequences can be extracted from different apps, i.e., opcode sequence duplication, the performance of the approaches mentioned above might be impacted.

Similar to opcode sequences, Android APIs have also been recurrently leveraged as features for machine learning-based Android malware analysis. For example, in 2013, Aafer et al. [3] have performed a thorough analysis that leverages critical API calls as features to evaluate the difference among selected classification algorithms. In their experiments, they employed four algorithms, including DT, C4.5, KNN, and linear SVM. Their experimental results reveal that KNN is the best algorithm for predicting malware when API calls are

considered as features. Similarly, in 2016, Wu et al. [169] leveraged the use of dataflow-related API-level features to improve the performance of a KNN detector. We observe that approaches of leveraging API calls as features may be impacted by API call duplication if the authors do not carefully sanitize their training dataset.

Machine learning-based Android malware family classification. In addition to machine learning-based malware detection, practitioners and researchers have also spent a significant amount of effort to identify the family of Android malware [52]. For example, Garcia et al. [52] proposed a novel approach for detecting malware families. By leveraging features extracted from specific Android API usages, reflective calls, and native binaries, they designed and implemented a prototype tool RevealDroid to achieve this purpose.

Most state-of-the-art approaches leverage unsupervised learning to identify Android malware families. The rationale behind this is that similar malware (belonging to the same family) will be grouped into the same cluster. As an example, Bayer et al. [19] have identified and grouped malware exhibiting similar behavior with a scalable clustering method. Similarly, in 2013, Hu et al. [61] designed and implemented a framework, namely MutantX-S, to cluster samples into families based on code instruction sequences efficiently. They have also proven that MutantX-S is highly accurate in detecting previously unknown malware. In 2015, Aresu et al. [12] created Android malware clusters by analyzing specific statistical information related to the HTTP traffic.

The above papers used clustering methods to aggregate malware with similar malicious behavior, which is of great significance for obtaining the family classification labels of malware. Unfortunately, none of these approaches has taken into account the sample duplication problem in their experimental setting, and thereby their performance might not be reliable.

Big code analysis for bias in machine learning. Apart from applying machine learning techniques to characterize Android malware, researchers have also started to investigate the potential biases that appear in the working processes of machine learning-based techniques. Pendlebury et al. [130] recently presented a study conducted on big code which discusses the potential biases in two dimensions: space (referred to as spatial bias) and time (referred to as temporal bias). Spatial bias is caused by the unrealistic setting of the ratio of benign to malware samples in training and test data. Temporal bias refers to the integration of future knowledge about test data into the training stage. Similarly, Li et al. [84] have experimentally shown that time inconsistency introduces significant biases to machine learning-based malware detection approaches, with the experiments based on big code.

In 2018, Li et al. [80] presented a study demonstrating that more features used by a machine learning approach do not necessarily mean better performance. In a recent work reported by Irolla et al. [66], 49.35% of the samples in the Drebin dataset have at least one more sample containing the same sequence of the opcode. This result is in line with the findings of this work. Indeed it actually motivated us to investigate the potential impact of such duplication on the performance of machine learning approaches.

Studies on the adverse effects of code duplication in machine-learning models have also been carried out. Allamanis et al. [6] presented a technical report describing the impact of multiple file-level (near-)clones appearing in large corpora of code. They discussed the biases introduced mathematically and empirically proved with big code analysis that code duplication can lead to overestimating performance when evaluating machine learning models.

In summary, previous works have proposed numerous machine learning-based approaches for detecting malicious apps, and various studies based on big code analysis have investigated potential biases that can arise in machine learning. However, no work has thoroughly examined the impact of sample duplication in big code on machine learning-based Android malware detection from different perspectives. In this thesis, we address this gap by performing extensive experiments to measure the effect of sample duplication on both supervised malware classification models and unsupervised learning models (e.g., malware family clustering) with big code analysis-based techniques.

2.2 Recommendations for Android Development

Recommender Tools in software engineering. Researchers have made many efforts to provide developers with various recommendation features as a primary recommendation for modern IDEs [126, 146, 127]. In recent years, concerning further improving the efficiency of developers, advanced research works based on recommendations are emerging [32, 154, 181, 18, 46, 118, 140, 116]. As another example, Gu et al. [53] present a graph kernel-based approach to the selection of API usage examples by representing source code as object usage graphs.

Mcmillan et al. [112, 113] utilize graph-based matching to retrieve and visualize associated functions and their usages. Chan et al. [30] propose an optimized algorithm to search in an API graph with the given text query phrases. Thung et al. [155] take as input a textual description of a feature request and recommend API methods based on the similarity between textual API descriptions. Rahman et al. [134] exploit the keyword-API association identified by crowd-sourced knowledge of StackOverflow to enrich the translation between natural language query and code search. Raghothaman et al. [133] propose a work to suggest code snippets given API-related natural language queries by learning structured API call sequences from open-source code repositories. Huang et al. [64] propose BIKER, which leverages both Stack Overflow posts and API documentation to extract candidate APIs for ranking with a programming task described in natural language. The above works frequently mention the utilization of text as query phrases. However, in this

thesis, the topic text of apps is uniquely used as a type of context together with code implementation, which is significantly different from the traditional modes of code search with text queries.

In light of the rise of large language models, a variety of strategies for automated API recommendation have emerged [166, 183]. These techniques can be broadly classified into two distinct paradigms: Information Retrieval-based (IR-based) and neural-based methods. Our study primarily focuses on the neural-based approaches. Wei et al. [166] introduced CLEAR, a sophisticated API recommendation system that employs BERT sentence embeddings to represent queries. This captures the continuous semantic aspects of the queries. Utilizing contrastive learning, CLEAR enables BERT to generate semantic representations that extend beyond mere lexical features. On a more recent note, Zhang et al. [183] presented ToolCoder, a novel approach that integrates API search capabilities with existing machine learning models for the dual purposes of code generation and API selection. This method employs an automated data annotation technique, which uses ChatGPT to enrich source code data with tool usage annotations. The code generation model is then fine-tuned based on this enhanced dataset. During the inference stage, an API search tool is cohesively embedded into the code generation pipeline, allowing the model to make informed API recommendations autonomously.

Collaborative filtering applied in software engineering. Collaborative filtering techniques are widely utilized in software engineering in general to implement recommendation systems. To assist developers in taking advantage of available third-party libraries, Thung et al. [153] combine association rule mining and user-based collaborative filtering and propose a technique to recommend likely relevant libraries according to the libraries an application currently uses. Furthermore, Yu et al. [176] introduce an approach that combines collaborative filtering and Latent Dirichlet Allocation (LDA) to provide sugges-

tions about third-party libraries for mobile apps. Moreover, He et al. [58] design a novel approach leveraging Matrix Factorization (a classic collaborative filtering based prediction approach) for predicting useful third-party libraries. Similarly, Xia et al. [171] employ an approach that combines a Matrix Factorization based latent factor model with a neighborhood-based method to capture implicit relations for improving the code reviewer recommendation, which has been acknowledged to be of great importance for software quality assurance. That is because due to the complexity of expertise and availability of candidate reviewers, it can be quite challenging to find appropriate reviewers.

Collaborative filtering techniques have also been introduced to recommend sampling methods to improve the performance of software defect prediction. For example, Sun et al. [149] present a collaborative filtering-based sampling method recommendation algorithm to automatically suggest appropriate sampling methods for newly identified defect data.

Big Code Augmented Recommendations for Android Development. Since the development process of mobile apps relies heavily on API frameworks and libraries, in the Android community researchers have devoted much effort to supporting recommendations by mining similar usages from big code to better support Android app development.

Some works try to give appropriate recommendation suggestions on third-party libraries [109, 97, 58], and others knuckle down to the code level, that is, giving real-time suggestions during app development. Hence, many works have been done to extract parameters as recommendations in similar programming scenarios [135, 86]. Except for recommending third-party libraries or parameter values for APIs, there are a large number of researchers focusing on recommending Android APIs and their usage patterns. According to Wu et al. [168]'s defined categories, we introduce state-of-the-art research works related to API recommendations. Most integrated development environments (IDEs) have been widely equipped with code completion features, which have been shown effective and

useful by developers in many studies. Through the basic recommendation features of IDEs, developers can promptly complete an API by typing 'dot' subsequent to an object instance to obtain a recommendation list generated based on the static information of the Android app under development.

At present, most of the advanced API recommendation mechanisms predict API usage patterns that are generated together with a given object instance. For example, Nguyen et al. [121] propose a sequence-based tool named DroidAssist to perform code completion for method calls based on the Hidden Markov model of API usages (HAPI). They subsequently provide another approach, the key component of which is also HAPI, to train a statistical model of API usage from Dalvik Virtual Machine (DVM) bytecodes [122]. The objective of these approaches is to recommend the next method call as well as a more suitable method sequence. Users are required to provide the object instances being edited as queries.

Clustering techniques have also been employed to object usage-based Android API recommendations. Niu et al. [124] mine API usage patterns by clustering the data based on the associations of object usages, i.e., API sequences on a given class, while building the "gold set" manually based on human programming knowledge is time-consuming and a potential threat to the construct validity in terms of both establishment and evaluation. In contrast, utilizes the context information to extract API usage patterns from similar apps, i.e., the construction is completed without human intervention. Yuan et al. [178] focused on the need to recommend event callbacks in the environment of Android application development and introduce an approach to recommend both functional APIs and the event callbacks that need to be overridden. The authors later extended their work by establishing Android-specific API description databases designating the associations among diverse functionalities and APIs [179].

Code search from big code has also been leveraged as a means to recommend API usages for Android developers. Indeed, with the objective of code search, Gu et al. [54]

introduce an approach to generate API usage sequences based on a natural language query through a deep learning-based approach. Similarly, Jiang et al. [69] generate recommended code snippets based on multi-aspect features, including text, topic, the number of lines, etc.

Different from the aforementioned code search methods, Nie et al. [123] propose an approach leveraging knowledge learned from Stack Overflow, which is also a popular source of big code, to grow the performance of code search algorithms. Nguyen et al. [119] introduce a context-aware collaborative filtering-based algorithm to recommend Java method invocations utilizing Rascal M^3 model [1]. Later, the authors extend their work by leveraging the algorithm to the Android platform [120].

In summary, big code analysis-based techniques have been widely applied to Android API recommendations. However, most of the state-of-the-art works are implemented based on techniques such as clustering, code search, or traditional predictive models that come with different limitations. This thesis focuses on providing recommendations for Android development by leveraging a context-aware collaborative filtering-based approach and extending the context to multi-dimensions to make full use of different features.

2.3 Compatibility Issues in Android Apps

We discuss two closely related works in addressing Android app compatibility issues from two aspects, i.e, compatibility analysis and program repair.

Compatibility Analysis. The issue of heavy fragmentation on the Android platform has resulted in a multitude of challenges for both app developers and users. With hundreds of manufacturers releasing various versions of Android, both official and customized, compatibility issues have become prevalent throughout the Android ecosystem. As a result, users frequently encounter frustrating experiences such as not being able to install certain apps, or encountering crashes when attempting to use certain functions. According to recent research conducted by Byron Muhlberg, there are now over a billion Android devices that

[1] https://www.eclipse.org/jdt/core/

are no longer supported by Google. The users of those devices will likely encounter compatibility issues, especially when they want to leverage the latest Android apps, leading to serious problems in the mobile ecosystem. Therefore, compatibility issues have been a key research topic in the Android community [164, 160, 145, 165, 63]. To assist developers in exhaustive app testing, Wei et al. [164] empirically study the fragmentation-induced issues to characterize the symptoms and root causes and propose a technique named FicFinder to detect such compatibility issues. After that, the authors [165] further present an API-device correlation extracting and learning approach named Pivot to help detect fragmentation-induced compatibility issues. Huang et al. [63] delve into the callback API evolution-induced compatibility issues and provide a technique named CIDER, leveraging a graph-based model to detect two types of callback compatibility issues. Unfortunately, both Pivot and CIDER focus on detecting certain types of incompatibility issues instead of repairing them, which constitutes the motivation of our research to some extent.

The exploration of compatibility issues caused by Android OS evolution is needed as apps are inseparably linked to the official Android APIs. Researchers have put a lot of effort into deprecated APIs [173, 92, 93, 57, 56, 44], which could eventually lead to compatibility issues. Li et al. [92] build a prototype tool called CDA and apply it to different revisions of the Android framework to characterize deprecated Android APIs. Based on an extensive empirical study, He et al. [57] reveal that drastic API changes exist between neighboring Android versions. They have additionally developed a tool named IctApiFinder to detect incompatible API usages. Similarly, Li et al. [91] propose an approach named CiD to model the lifecycle of the Android APIs and flag the error usages capable of causing compatibility issues, the issues declared by which are also regarded as one of our motivations. Xia et al. [170] perform a large-scale study on the practice of handling OS-induced API compatibility issues and their solutions, and they propose a tool called RAPID to ascertain whether a compatibility issue has been addressed.

In non-Android communities, research on API deprecated is also ubiquitous [137, 60, 188, 22, 144]. Zhou et al. [188] study API deprecation usage in open-source Java frameworks and libraries. They also propose a framework to detect deprecated API usages in source code examples on the Web. Brito et al. [22] perform a large-scale analysis of real-world Java systems and reveal that there is almost no significant effort to improve deprecation messages. Some researchers concentrate on the impact of API deprecation [60, 137, 144, 33]. Hora et al. [60] report on an exploratory study that aims to observe API evolution and its impact on the Pharo ecosystem. Sawant et al. [144] extend the study on Java and investigate how many API clients update their dependencies to actively maintain their projects and count the number of affected projects by deprecation.

Program Repair. Program transformations and repairing have been widely researched [23, 67, 70, 78, 139, 114, 159, 34]. For instance, LASE [67, 114] is an example-based program repair tool by learning non-trivial data and its context from multiple editing examples and automatically searching for editing locations to apply customized editing to these locations. Rolim et al. [139] present REFAZER, a technique based on the code edits performed by developers for automatically learning program transformations. Coccinelle [23, 78] is a C-based program matching and source-to-source transformation tool that has been employed for the automatic evolution of the Linux kernel. Coccinelle provides Semantic Patch Language (SmPL) to write its transformation rules. As a Java extension to Coccinelle, Coccinelle4J [70] is designed to apply to Java programs. Similar to Coccinelle, it uses semantic patches written in SmPL.

Recently, studies have begun to focus on the usage of automatically updating incompatible Android APIs. As one of the first tools to implement this goal, AppEvolve [44] uses GitHub as the code base to perform API updates by learning examples before and after the update. Haryono et al. [56] improve AppEvolve by proposing CocciEvolve, which uses

a single updated example to perform API updates and provides readable and configurable scripts in the form of semantic templates. They further broaden their study by proposing AndroEvolve [55], which addresses the defects of CocciEvolve with data flow analysis and variable name denormalization. Lamothe et al. [77] leverage the basic *diff* in the version control system to learn API migration patterns, where they use the ASTs to match the API calls in the source code to the code examples.

In summary, most of the previous works only aim at detecting compatibility issues. In terms of compatibility issue fixing, all current approaches focus on repairing the source code of Android apps and hence cannot be applied to directly repair published Android apps. As a supplement to existing repair approaches that attempt to help developers in developing higher-quality apps, we believe there is also a need to provide approaches for helping repair published apps. This thesis focuses on leveraging pre-defined patch templates which are created based on the knowledge learned through big code analysis to instrument published Android apps to fix compatibility issues.

Chapter 3

On the Impact of Sample Duplication in Machine Learning based Android Malware Detection

Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F. Bissyandé, Jacques Klein, and John Grundy. 2021. On the Impact of Sample Duplication in Machine-Learning-Based Android Malware Detection. ACM Trans. Softw. Eng. Methodol. 30, 3, Article 40 (July 2021), 38 pages. https://doi.org/10.1145/3446905

3.1 Introduction

Security of mobile apps is now a critical research and practice issue. Mobile apps are used pervasively, including for critical activities such as transportation (e.g., smart car apps), finance (e.g., banking apps), and healthcare (e.g., heart rate monitoring apps). Even leisure apps, which are seldom viewed as critical, may pose security threats given that they can provide attackers easy access to sensitive information on users' devices [15, 83]. The diversification of app developers, vendors, and brokers creates a great challenge to ensure that each app can be rapidly and effectively vetted from the perspective of security and privacy concerns.

Ideally, mobile apps should be intensively analyzed to check their security and privacy requirements conformance. However, when performed statically, app analysis is often time-consuming, may produce many false positives, and will not identify all problems

that occur at runtime [90, 50, 125, 88, 148]. On the other hand, dynamic analysis does not scale and often cannot cover the whole codebase [73, 108, 39, 107]. In recent years the research community has progressively shifted to view machine learning (ML) as an affordable and worthwhile effort for identifying security issues, in particular, malicious behavior, in mobile apps [85, 62, 51, 96].

An increasingly large body of research on machine learning-based approaches for predicting Android malware has been published. DREBIN [13] and MamaDroid [110] are state-of-the-art sample approaches that are commonly referred to. However, after promising results have been reported, the community has started to reflect on the potential biases that many machine learning-based research experiments carry. For example, Allix et al. [7] and then Pendlebury et al. [130] have experimentally shown that the performance of malware detectors is actually highly dependent on experimental parameters, such as dataset construction (e.g., spatial and temporal dimensions) or evaluation methodology (e.g., 10-fold cross validation). Dataset is a critical ingredient in the training and validation of all machine learning-based models. Nevertheless, to date the app analysis community has paid little attention to the intrinsic quality of datasets beyond the problems of class imbalance and temporal alignment [7, 130].

Concerning the quality of datasets and their impact on machine learning models, Allamanis [6] has recently raised the concern of *code sample duplication*, i.e., the phenomenon where a given sample is repeated several times in the dataset. This study reported that performance metrics of machine learning models for big code are sometimes inflated by up to 100% when testing on duplicated code corpora, compared to their performance on de-duplicated corpora. We consider this alarming finding to be relevant for further investigation in the field of ML-based Android malware detection since Android samples may also contain duplicated features, which are often extracted from Android apps' code snippets and metadata such as permissions or resource files. Indeed, it casts doubts on threats to

validity on all research achievements in this area. Our work, in this paper, echoes this concern and attempts to clarify the effect of sample duplication on the performance of malware detectors.

Our investigations start with a review of recent state-of-the-art approaches in ML-based malware detection. Through this we identify some artifacts whose duplications in the datasets may not be obvious to experimenters. We then discuss theoretically the possible effect of sample duplication on learning models. Then, we quantify the extent of sample duplication in widely-used app malware datasets. Finally, we perform large scale experimental analyses of the effect of duplication by considering two learning scenarios for malware detection – binary classification of maliciousness as a supervised learning problem, and malware family clustering as an unsupervised learning problem. Based on our experiment findings, we provide a discussion on the quality of the current malware datasets as well as on the validity of their recorded performance in the literature.

Our experimental exploration reveals that (1) widely-used malware datasets include a considerable amount of duplicate data samples. Given recent studies on the adverse effect of duplicate code for machine learning models of code, many promising results and findings in the literature are potentially threatened by this issue of duplication bias. With comprehensive experiments covering different types of dataset samples, we have shown that (2) the impact of duplicates in commonly-used datasets remains marginal. This is for the typical supervised learning models that are proposed for app malware detection, no matter 10-fold cross-validation or in-the-wild experiments are applied. (3) the marginal impact is consistent across different machine learning models trained on different algorithms including RandomForest and SVM. (4) We have shown that unlike supervised learning for which insignificant impact is observed for malware detection, the impact of sample duplication on unsupervised learning (especially on malware clustering) is however quite significant. Based on these empirical findings, no matter how significant the impact of sample duplication might be, we appeal to the community that our fellow researchers and practitioners should always take sample duplication into consideration when performing future machine learning (via either supervised or unsupervised learning) based Android malware detections.

The remainder of this paper is organized as follows. Section 3.2 explains the problem scope of this work and Section 3.3 presents a preliminary study regarding the duplica-

tion phenomenon in publicly released Android malware datasets. After that, Section 3.4 presents our experimental design while Section 3.5 reports the corresponding experimental results. Later, Section 3.6 discusses the possible implications of this work and its potential limitations. Section 3.7 discusses the related work and finally Section 3.8 concludes this paper.

3.2 Problem Scoping

We provide key background information for helping readers better understand our study. Notably, we recall the main usage scenarios of machine learning in the field of malware detection and discuss different levels of app details that are recurrently leveraged to constitute learning datasets. Finally, we introduce the duplication bias problem.

3.2.1 Machine Learning based Malware Detection

Machine learning is relied upon to perform malware detection at a large scale in many published toolsets and experiments. There are mainly two scenarios: *binary classification* models are trained to predict the maliciousness of an app. These are generally a *supervised learning* scenario where the entire dataset is labeled for the experiments. In contrast, the problem of malware family identification is often modeled as an *unsupervised learning* scenario, i.e., samples are grouped together based on their similarity.

Table 3.1 enumerates a few examples of key published work in the field of machine learning based malicious behaviour analysis. A key observation from this table is that for the large majority of approaches, feature engineering targets **code** artifacts. Our study will thus focus on code-related samples.

For the purpose of our study, we focus on four levels of artifacts that may be subject to duplication when they constitute the learning datasets: (1) the **APK** samples (i.e., the whole app packages), (2) the **DEX** code (i.e., the entire code within the app package), (3)

Table 3.1: Sample approaches involving machine learning for malicious Android apps analyses.

Year	Venue	Approach	features	Detection scenario
2019	TASE	CDGDroid [186]	CFG, DFG	Family classification
2018	TOSEM	RevealDroid [52]	API calls, reflection, native binaries	Bi- and family classification
2017	CODASPY	McLaughlin et al. [111]	Opcode sequence	Bi-classification
2016	NDSS	MaMaDroid [110]	API calls	Bi-classification
2016	IST	Wu et al. [169]	API calls	Bi-classification
2015	ICSE	MUDFLOW [17]	Sensitive data flows	One-class classification
2015	ICSE	Holland et al. [59]	API calls, permissions	Bi-classification
2014	NDSS	DREBIN [13]	API calls, permissions, etc.	Bi-classification
2014	SIGCOMM	Droid-Sec [180]	API calls, permissions, dynamic behavior	Bi-classification
2013	SecureComm	DroidAPIMiner [3]	API calls	Bi-classification

the **Opcode Sequence** (i.e., the low-level machine language instructions), and (4) the **API calls** (i.e., only the specific parts of the code that interact with the framework and access sensitive physical resources).

In practice, datasets used for learning are composed of samples of each of the aforementioned types and can include duplicates. This may subsequently lead to duplication bias (cf. Section 3.2.2).

APK Android apps are distributed in the form of packages (APKs). Every app version has a distinct APK file. Malware datasets shared in the literature are generally formed by collecting APKs from a variety of sources such as the official Google Play store and other alternative third-party markets. The state-of-the-art machine learning based malware classifiers are usually proposed to flag Android malware at the APK level.

DEX DEX file (i.e., Dalvik Executable) is the core file of an Android app that contains the actual programming code to be executed on a hosting Android device OS. By default, the DEX file is named as *classes.dex* in given Android apps. In this work, we consider DEX duplication exists in a dataset as long as the same DEX file appears in different apps (i.e., the hash value of the *classes.dex* file is identical between two different apps). For example, in the Drebin dataset, there are 128 different app samples that share the same

package name, namely com.soft.android.appinstaller, for which their DEX hashes are the same despite their APK hashes being different.

Opcode Sequence Bilar et al. [20] assert that opcodes (which can be disassembled from DEX files) can act as a predictor, since the distribution of malware opcode frequencies significantly differs from that of non-malicious software. In our manual observation, we found that different DEX files can indeed result in identical opcode sequences. These can be fulfilled by, for example, altering only the resource files of given Android apps, where some resources files in Android are used to set the constant values of certain variables or define the names of specific widgets. This type of change will lead to different DEX files but will not impact the actual code compiled to DEX files. Thereby, they will not impact the disassembled opcode sequences. In this work, we consider opcode sequence duplication to exist as long as different apps share the same opcode sequence. This is no matter whether the corresponding DEX files are identical or not. As an example, there are three app samples, namely com.brianrileyar.girlonfire, com.brianrileyar.fieldrunners, and com.brianrileyar.sonic, sharing the same opcode sequences but with different DEX hashes.

API Call We go one step further to identify more fine-grained duplications in malware datasets. To this end, we look at the API calls (which can be extracted from an app's opcode) of Android apps, since APIs are one of the most important constituent parts of the apps. Even if two apps have different opcode sequences, these two apps may still have the same sequence of API calls, which may in turn lead to identical features when APIs are exclusively considered. In this work, we consider API call duplication to exist as long as there are a number of apps that access into the same number of APIs, and each API is called in the identical sequence and times. For example, in the Drebin dataset, the samples with package names, com.evilsunflower.reader.evilShenger, com.evilsunflower.reader.evilQichang, com.evilsunflower.reader.evilGuigu, and com.evilsunflower.reader.evilLiangxing, share the

same API calls, although their opcode sequences are different.

3.2.2 Duplication in Machine Learning Datasets

Machine learning experiments require datasets to train models that provide recommendations on *new* and *unseen* samples. The standard expectation is that the models will *generalize* well to those new samples: the training process faithfully models the true distribution of the data as it will be observed by the particular application execution scenario. As discussed by Allamanis [6], in order for the machine learning model to generalize to the true data distribution[1], it needs to be trained on data independently drawn from that distribution. Unfortunately, sample duplicates commonly violate this assumption with varying consequences as duplicated samples will not introduce new knowledge to the learning models.

Sample duplication, in the domain of malware detection, refers to the idea that some learning data samples (e.g., the APK, the DEX code, etc.) appear multiple times within a corpus. The feature vectors subsequently extracted from the duplicated samples will likely be duplicated as well. Such duplication creates an issue as it biases the data distribution. Actually, a common practice in machine learning experiments is to split any existing dataset into two parts: a training set that is used to train the machine learning model and a test set where the performance of the model is measured. Since duplicated datasets are randomly distributed to the training set, the algorithms tend to learn different probability distributions, which may result in different results. Moreover, the splitting process may put the same samples (e.g., duplicated ones) into both training set and test set, leading also to biased learning.

Definitions: Assume a dataset D of app information samples that is split into a training and a test set. Conceptually, we can distinguish three types of duplicates: (1) "in-train" duplicates, i.e., samples duplicated within the training set; (2) "in-test" duplicates, i.e.,

[1]True data distribution is different from real-world data distribution since the former one should contain no duplicated samples while the latter one could.

duplicates within the test set; and (3) "cross-set" duplicates, i.e., samples that appear both in training and test sets.

We borrow the terms of Allamanis to define the **Duplicate bias** [6]:

In machine learning, a measured quantity f, such as the loss function minimized during training or a performance (e.g., precision) metric, is usually estimated as the average of the metric computed uniformly over the training or test set(s). Specifically, the estimate of f over a dataset $D = \{x_i\}$ is computed as

$$\widehat{f} = \frac{1}{|D|} \sum_{x_i \in D} f(x_i) \tag{3.1}$$

Duplication biases this estimate because some wx_i will appear multiple times. Specifically, we can equivalently transform D as a multiset $X = \{(x_i, c_i)\}$ where $c_i \in \mathbb{N}+$ is the number of times that the sample x_i is found in the dataset. Therefore, we can rewrite Equation 3.1 as

$$\widehat{f} = (1-d) \underbrace{\frac{1}{|X|} \sum_{x_i \in X} f(x_i)}_{\text{unbiased estimate } \bar{f}} + d \underbrace{\frac{1}{|D|-|X|} \sum_{x_i \in X} (c_i - 1) f(x_i)}_{\text{duplication bias } \beta} \tag{3.2}$$

where $d = \frac{|D|-|X|}{D} = \frac{\sum c_i - |X|}{D}$ is the *duplication factor* where $|X|$ is the number of unique x_i in X. Thus d is the proportion of the samples in the dataset that are duplicated ($c_i > 1$). By rewriting the above equation as $\widehat{f} = (1-d)\bar{f} + d\beta$ we see that the larger the duplicate factor d, the larger the effect of the duplication bias β.

From a machine learning perspective, the duplication bias in the training loss causes a model to overweight some training samples (the in-train duplicates). During testing, the duplication bias will skew the reported performance metric. Furthermore, we expect cross-set duplicates to artificially improve any metric taking advantage of the fact that multiple samples that are seen during training also appear in the test set, giving the illusion that the model generalizes, where in fact it memorized duplicates.

3.3 Published Datasets for Malware Detection

Our study aims at uncovering potential issues of stat-of-the-art ML-based malware detection approaches due to duplication bias. To this end, we first focus on investigating the presence of duplicates within commonly-used Android malware datasets in the described in the literature. Towards investigating the impact of sample duplication in ML-based Android malware detection, we are interested in knowing, in the first place, if sample duplication indeed exists in common Android Malware datasets. Our research question is thus:

RQ1: Does the duplication phenomenon, which has been revealed in big code modeling datasets, occur in Android malware datasets?

3.3.1 Study Datasets

Android Malware Detection and Analysis has received much attention for many years. The research community has collected and updated from various sources a variety of datasets that provide a ground truth for evaluating technical approaches to app malware analysis. Table 3.2 summarizes four representative ones[2] and provides some descriptive statistics about their size and diversity in terms of numbers of malware families that are represented. We then give a brief introduction to these four exemplar datasets.

Table 3.2: Statistics of Selected Android malware datasets.

Dataset	#. Malware	Collecting Period	Release Date	#. Families
Genome	1,260	2010 $\rightarrow$ 2011	2012	49
Drebin	5,560	2010 $\rightarrow$ 2012	2014	179$^{\alpha}$
AMD	24,553	2010 $\rightarrow$ 2016	2017	71
RmvDroid	9,133	2014 $\rightarrow$ 2018	2019	56

$^{\alpha}$ adware is excluded from this dataset.

- **Genome.** The Genome project is a seminal work in the research of Android mal-

[2]These datasets have been widely used by our community to evaluate the effectiveness of malware detection and classification approaches [161].

ware detection. As part of this project, Zhou et al. [189] publicly released a dataset of 1,260 malicious apps covering the majority of existing Android malware families (specifically, 49 families that were manually labeled by security analysts). The release dates of the app samples range from August 2010 to October 2011. Nowadays, Genome is considered to be obsolete, as most malware signatures have been well understood and malware writers are devising new techniques to hide malicious behavior (both from static checkers and dynamic monitoring).

- **Drebin.** To foster research on Android malware and to enable comparison among different detection approaches, Arp et al. [13] released the Drebin Android malware dataset in 2014, for which they built as part of their work on "explainable malware detection". The Drebin dataset contains 5,560 malicious apps that are collected between August 2010 and October 2012. This dataset also includes the samples from the Genome dataset. The 5,560 malicious apps are categorized by Drebin maintainers into 179 families. Unlike the families labeled in the Genome project, which are labeled mainly by practitioners, the malware samples in the Drebin dataset are labeled by the authors of the Drebin approach themselves based on the output of different anti-virus scanners. The authors took steps to manually unify the output of these anti-virus scanners.

- **AMD.** AMD is a carefully-labeled and well-studied Android malware dataset [163]. In addition to 24,553 samples assembled from 2010 to 2016, the dataset also includes a manually documented behavioral description of its malware samples. Based on the results of anti-virus scanners, the malware samples of this dataset are categorized into 135 variants within 71 malware families.

- **RmvDroid.** Released in 2019, RmvDroid [161] is the latest malware dataset that was released for complementing existing datasets, which are often outdated, unreliable,

and lacking details of app metadata such as description, reviews, etc. The RmvDroid dataset contains 9,133 app samples that are associated with 56 malware families and were all caught after being exposed in the official Android market (i.e., Google Play).

Table 3.3 further enumerates some representative state-of-the-art approaches that have leveraged these datasets to train malware classification models. The last three columns of this table further illustrate the performance (i.e., precision, recall, and F1 score, respectively) achieved by those approaches. The fact that all the approaches have achieved over 97% F1 score (or 96% precision, 95% recall) suggests that these datasets selected in this work are representative and hence are suitable to fulfil our experiments.

Table 3.3: The performance achieved by some representative state-of-the-art approaches that have leveraged these datasets to train malware classification models.

Dataset	Approach	Precision(%)	Recall(%)	F1 score(%)
Genome	Fan et al. [43]	99.67	95.85	97.72
Drebin	DANdroid [115]	98.4	98.9	98.6
AMD	Li et al. [79]	99.22	99.16	99.19
RmvDroid	Fan et al. [43]	96.54	97.77	97.15

3.3.2 Duplication in Malware Datasets

An Android app is identified in the official market based on its unique package name, which prevents users from installing different versions of a given app on their device. However, since malicious code can be inserted during app updates, app versions may be considered as different samples within the real-world distribution of apps. Therefore, maintainers of datasets generally rely on hash values of APK files to ensure that sample APKs are unique.

Cross-dataset APK duplication Although commonly-used datasets do not include duplicated APKs (the samples are often named by their SHA256 hash value), we note that the datasets are often redundant from one to another. This redundancy should be made

clear to researchers who are interested in combining multiple datasets to fulfil their experiments [174, 43]. We hence provide in Table 3.4 statistics of cross-dataset duplications. We note that APK redundancy exists in all the considered malware datasets. The duplication rate varies from as small as 1% to as large as 97%, although those malware datasets are all collected via different methods. Even though Drebin authors have claimed that it included all the Genome samples, due to some outlier cases, we cannot observe a 100% duplication rate for these two datasets.

Table 3.4: APK duplication between Selected Android malware datasets. The duplication rate is calculated via the following formula: $\frac{|Dataset1 \cap Dataset2|}{min(|Dataset1|,|Dataset2|)}$.

Dataset1	Dataset2	#. Cross	Duplication Rate	Dataset1	Dataset2	#. Cross	Duplication Rate
AMD	RmvDroid	2,618	28.67%	AMD	Genome	365	28.97%
Drebin	Genome	1,229	97.54%	Drebin	RmvDroid	40	0.72%
AMD	Drebin	559	10.05%	Genome	RmvDroid	26	2.06%

Within-dataset duplication While researchers appear to ensure that datasets are not duplicated at the APK level, we note that the relevant in-APK components may be duplicated. For example, including several repackaged versions of a given app, where only layout and image changes are applied, will lead to a duplicated dataset of code (Dex) clones. Similarly, looking at lower-level details that constitute the samples for learning, one might discover new duplications that prevent to faithfully model the distribution of data that practitioners have to deal with. Fig. 3.1 summarizes the statistics of duplication for dex code, opcode sequence, and API calls in all the four selected datasets.

We note that Dex code duplication concerns between 2.6% of samples (the RmvDroid dataset) and 40% of samples (the Drebin dataset). For the case of the Drebin dataset, this means that for 40% of its app samples, there exists at least one other sample in the dataset that shares the same DEX file with them. Although the percentages are substantial, they are far less than the percentages of duplication for Opcode sequences (minimum of

30%) and API calls (minimum of 40%). Distribution of the number of samples in each duplication are further provided in Fig. 3.2 to provide more insights. The median values of all the distributions[3] shows that at least half of the duplications happen on only two samples, indicating the selected datasets are quite diverse despite the existence of sample duplication. The fact that the maximum values of all the distributions are always less than five also backs up this indication.

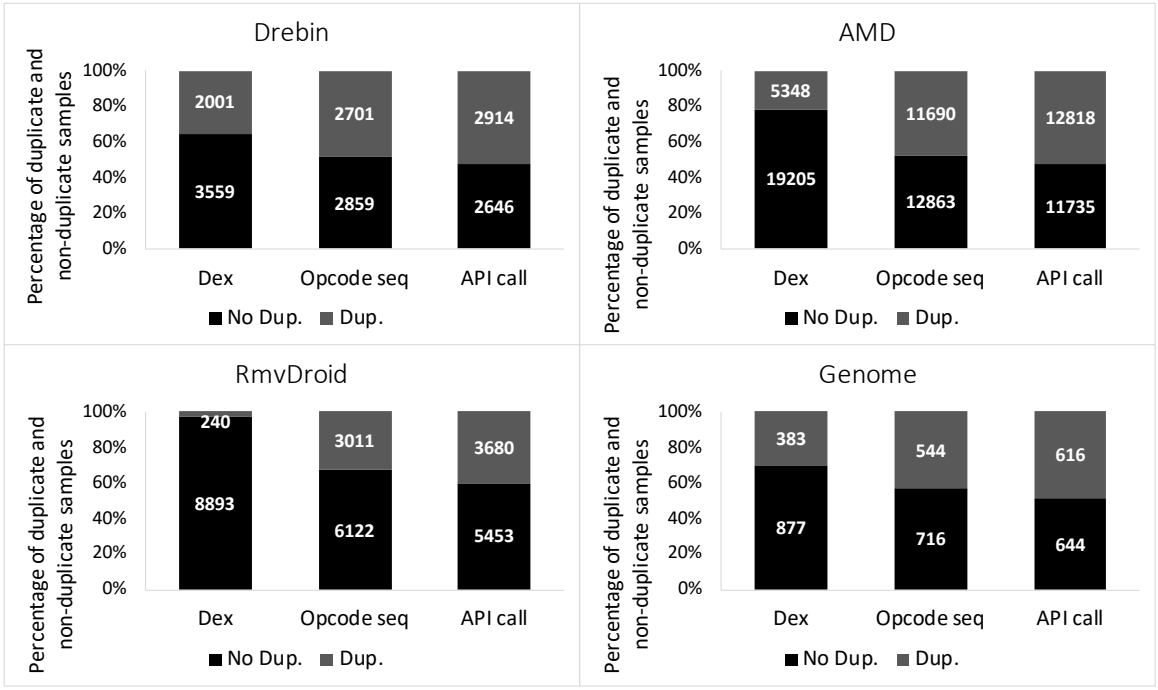


Figure 3.1: Within-dataset sample duplication in the selected malware datasets.

Table 3.5: Malware Family intersection between selected Android malware datasets. The intersection rate is calculated via the following formula: $\frac{|Dataset1 \cap Dataset2|}{min(|Dataset1|,|Dataset2|)}$.

Dataset1	Dataset2	#. Same Families	Intersection Rate	Dataset1	Dataset2	#. Same Families	Intersection Rate
AMD	RmvDroid	12	21.43%	AMD	Genome	5	10.87%
Drebin	Genome	18	39.13%	Drebin	RmvDroid	10	17.86%
AMD	Drebin	21	29.58%	Genome	RmvDroid	5	10.87%

[3]The only difference is *Dex in the RmvDroid dataset, for which the median value is also quite close to 2.*

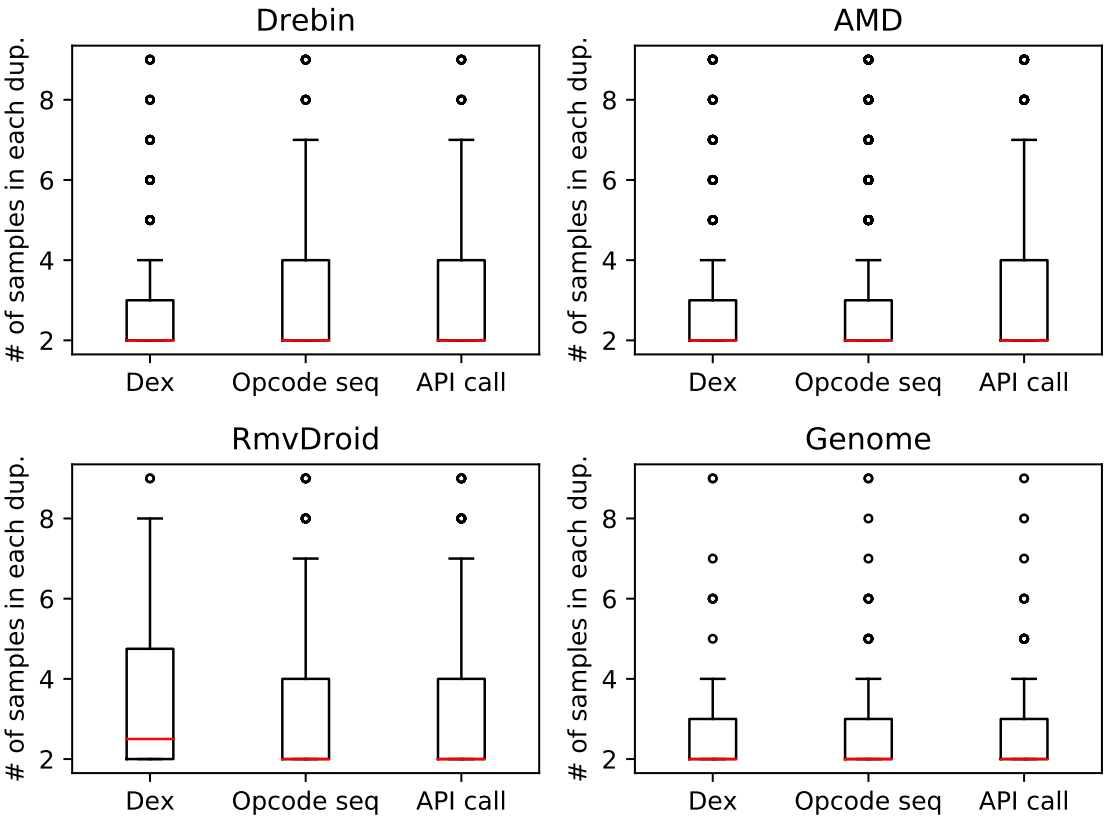


Figure 3.2: Distribution of the number of samples in each duplication (Samples without duplication are ignored).

3.3.3 Family representation in datasets (a.k.a. family duplication)

During our analyses, we have found that although datasets include samples from a large variety of malware families, the intersections of families between the selected datasets are relatively small. As summarized in Table 3.5, the intersection rate ranges from 10.87% to 39.13%. The actual distribution of the size of malware families is often highly imbalanced. For instance, in the Drebin dataset, within the 179 families that are enumerated, some are represented with a single APK sample, while other families include hundreds of samples, as illustrated in Fig. 3.3. This imbalance, if ignored, may introduce biases to malware clustering approaches and hence the performance of ML-based malware classifications. Indeed,

as argued by Yu et al. [177], in practice, samples in a dataset may have different importance when conducting clustering-based approaches. Therefore, it is important to properly adjust the sample distributions (or weights) when clustering a dataset. Unfortunately, the same phenomenon occurs for RmvDroid, AMD, and Genome datasets. For the sake of characterizing the imbalance among various families, we counted the malware samples from the top 10 families of each dataset and made a comparison with the whole dataset, as shown in Fig 3.4. For all the malware datasets, the top-10 family samples account for over 70% of the total samples. As shown in Table 3.6, samples in some malware families may have been highly duplicated (could be over 90%) as Android malware developers tend to create new malware by repackaging existing ones.

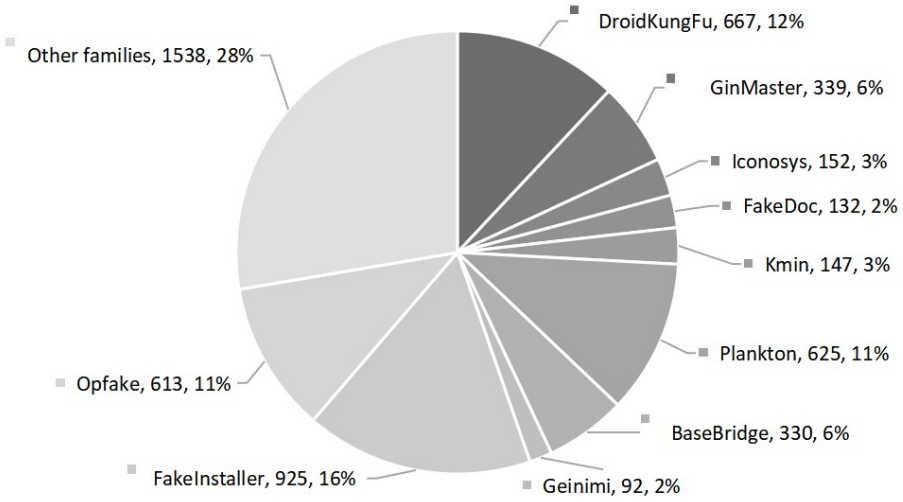


Figure 3.3: Family distribution of the Drebin dataset.

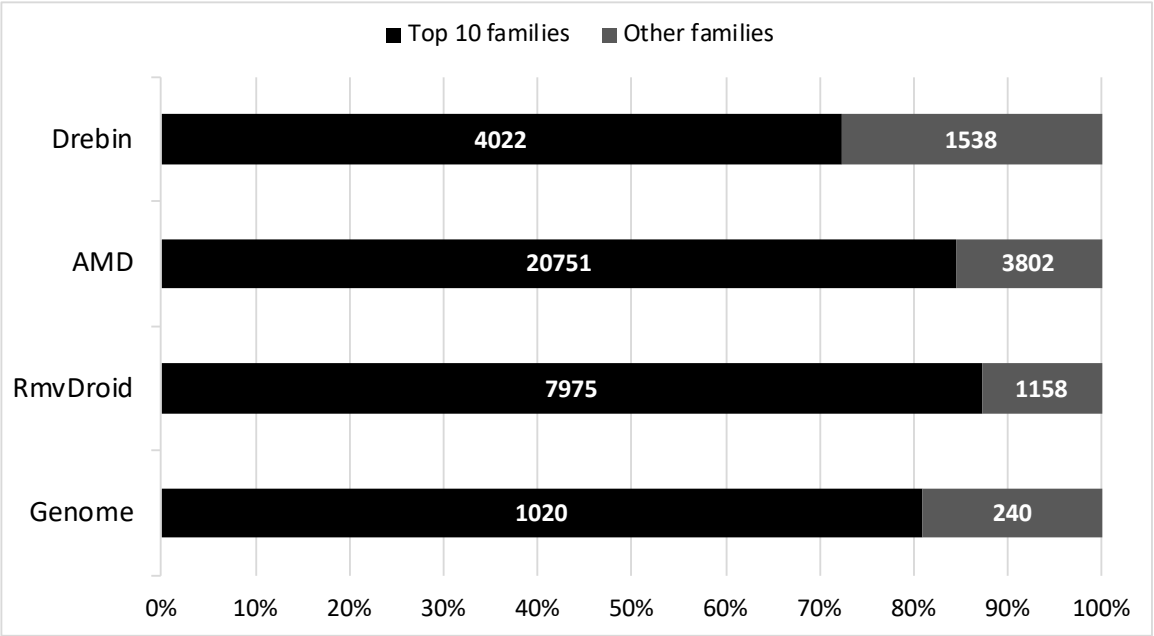


Figure 3.4: Imbalanced distribution of malware families in the selected four datasets.

Table 3.6: Sample duplication in the top-10 malware families of the selected Android datasets.

Drebin				AMD				RmvDroid			
Family	**Dex**	**Opcode seq**	**API call**	**Family**	**Dex**	**Opcode seq**	**API call**	**Family**	**Dex**	**Opcode seq**	**API call**
FakeInstaller	82.27%	84.86%	85.62%	Airpush	1.05%	36.23%	42.97%	Airpush	2.37%	43.28%	48.93%
DroidKungFu	18%	34.93%	44.98%	Dowgin	4.73%	22.62%	26.2%	Mecor	0	88.83%	91.48%
Plankton	0.32%	18.72%	22.88%	FakeInst	82.7%	85.29%	88.38%	Plankton	0	12.59%	19.88%
Opfake	80.91%	86.79%	87.11%	Mecor	0.27%	97.58%	97.91%	Adwo	0.28%	10.64%	39.5%
GinMaster	2.95%	3.24%	3.24%	Youmi	0.69%	16.94%	23.94%	Youmi	0	11.35%	21.31%
BaseBridge	60.91%	74.55%	77.58%	Fusob	89.83%	89.83%	94.25%	Mobidash	30.97%	78.13%	82.39%
Iconosys	0	41.45%	50.66%	Kuguo	1.58%	21.85%	25.77%	Kuguo	0.26%	3.6%	4.63%
Kmin	42.18%	67.35%	70.07%	BankBot	58.91%	80.87%	83.7%	Gappusin	0.65%	6.1%	18.74%
FakeDoc	68.18%	69.7%	69.7%	Jisut	72.71%	85.35%	91.21%	Viser	0.34%	9.52%	13.27%
Geinimi	15.22%	21.74%	26.09%	DroidKungFu	18.68%	33.88%	45.05%	Dowgin	2.38%	7.48%	9.18%

RQ-1 Answer

Duplication is commonplace in malware datasets used in the published literature. It occurs for samples at a different artefact level when leveraged for machine learning based malware detection. Duplication may concern up to 40% of Dex code samples in datasets that are widely used for experimental validation of detection techniques, or up to 97% in certain malware families, which may significantly bias evaluation results.

3.4 Study Design

We now present the detailed design of our empirical study, including the research questions we aim to answer (Subsection 3.4.1), the machine learning algorithms we will leverage in this work (Subsection 3.4.2), the features we plan to extract from Android apps (Subsection 3.4.3), and the evaluation metrics we will leverage to categorize the performance of machine learning classifications (Subsection 3.4.4).

3.4.1 Research Questions

Machine learning techniques for malware detection have been largely assessed using common datasets that include duplicated artefacts. We conduct several experiments to assess for any potential duplication bias, specifically the extent of sample duplication impact on the performance of state of the art ML-based Android malware detectors. To that end, we propose three additional refined research questions that explore cases of both supervised and unsupervised learning approaches.

- **RQ2:** What is the impact of malware sample duplication on supervised learning for building Android malware detectors?
- **RQ3:** Is the impact of sample duplication influenced by a specific underlying supervised learning algorithm?
- **RQ4:** Are unsupervised learning models impacted by sample duplication bias in a similar way to supervised learning models?

3.4.2 Machine Learning Algorithms

We now present the machine learning algorithms that are leveraged in this work to implement Android malware predictors. Using several state-of-the-art malware detection approaches described in the literature, we train binary classification models to predict mali-

cious content of sample apps. Specifically, we have focused on the following four algorithms. We first leverage SVM to answer RQ2 (cf. Section 3.5.1) and then leverage all the four algorithms to empirically compare the impact of duplication bias on different machine learning algorithms (cf. Section 3.5.2).

- **Support Vector Machine (SVM)** is a widely used supervised machine learning algorithm that attempts to perform binary classifications by finding the hyperplane that effectively separates data points associated with two classes [25]. In malware detection tools, SVM has been one of the key long term algorithms that have been explored. Even state of the art approaches such as Drebin have achieved high performance by relying on SVM [13].

- **Decision Trees (DT)** infers classification rules from a set of unordered and irregular cases. It performs classification using branch structure, using a tree as a form of expression. According to the survey of Safavian et al. [141], decision trees have been successfully used in many diverse areas. Aung et al. [16] have used decision trees to classify Android applications as malicious or benign by focusing on their permission features.

- The **K-Nearest Neighbor (KNN)** algorithm identifies the closest K instances (from the training dataset) based on a certain distance metric and from which it picks up the most common class tag among to form the prediction result. The study of Firdausi et al. [45] shows that simple machine learning algorithms such as KNN can be used to detect malicious applications effectively and efficiently.

- **Random Forest (RF)** is a classification algorithm that builds multiple decision trees from which the final output is converged by voting the results yielded by individual trees. Several tools for Android app analysis have used RF for malware classification. For example, Alam et al. [5] have largely relied on RF in their study. Sanz et al. [143]

have found that RF produces the best classifier among all the algorithms including Sequential Minimal Optimization (SMO) [35], KNN, etc.

3.4.3 Feature engineering

Machine learning-based classification relies on data to learn what characteristics could suggest that a given sample likely belongs to a given class. For example, in malware detection, the learning algorithms must be "told" what characteristics are associated with each malicious or benign sample in the dataset. Such characteristics are known as the feature set and implemented as a feature vector. Arp et al. [13] proposed one of the most comprehensive feature sets for Android malware detection, which has proved effective in the state of the art Drebin approach. They focused on features that can be extracted with a lightweight static analysis approach in order to scale their tool to thousands of samples. Such features are based on the Manifest file in the apk (i.e., *AndroidManifest.xml*) as well as the disassembled DEX code. In the end, eight (8) aspects are considered to produce 8 sets of strings:

- **S1: Hardware components.** Malicious behavior in Android malware often involves access to specific hardware components such as the camera or the GPS. Related features statically extracted based on access requests made can be derived from the manifest file.

- **S2: Requested permissions.** Malicious apps tend to request specific permissions more frequently, such as SEND_SMS, CAMERA, READ_CONTACTS than that of benign apps [143]. Hence permission requests from the Manifest file could be a good indicator to differentiate malware from goodware.

- **S3: App components.** Activities, services, content providers and broadcast receivers that are declared in the manifest file are the four types of existing components that define different specific interfaces to the system. Some components may be statistically

more leveraged by malicious apps than by benign apps.

- **S4: Filtered intents.** An Intent is a messaging object used to exchange data between app components. Malware may use it to listen to particular intents to achieve malicious purposes, e.g., malware could hijack the SMS_RECEIVED system intent to listen to users' SMS messages.

- **S5: Restricted API calls.** A typical case of identifying malicious behavior is the use of restricted API calls without requesting required permissions, i.e., the application has a high probability of gaining external permissions through an exploit of privilege escalation vulnerability. Such restricted critical calls can be detected from the disassembled code.

- **S6: Used permissions.** After harvesting the restricted API calls (as shown in S5), one can calculate the actual permissions required by the app accessed those APIs. This permission set might be different from the one explicitly declared by app developers in the manifest (cf. S2).

- **S7: Suspicious API calls.** Certain API calls may cause exposure to sensitive data or resources and are often exploited by Android malware. API calls for sensitive data access, network communication, sending and receiving SMS messages, executing external commands and frequently used for obfuscation are gathered.

- **S8: Network addresses.** In many cases, malicious apps eventually need to fetch external data (e.g., download dynamically loaded code) or to leak data outside the app. These activities require network communication with specific hosts. Thus, IP addresses, hostnames and URLs gathered from the disassembled code can be characterized for malicious behavior prediction.

The aforementioned feature set covers a wide range of characteristics of Android apps

and has been adopted widely by the research community [3, 111, 52, 41]. Our experiments in this work directly leverage this comprehensive feature set to build relevant classifiers that match the typical performance recorded in the literature. Consecutively, our study focuses on the impact of sample duplication bias on these classifiers. We note that the eventual size of the feature set in each experiment is dependent on the app dataset selected for training.

3.4.4 Evaluation Metrics

For a binary classification problem, the ML model ultimately needs to predict whether a given sample belongs to one of two classes (i.e., generally is positive or negative). In our case, we consider the classes *malware* and *goodware*. A confusion matrix is often used to establish the performance of a classification model, based on four measurements:

(1) True Positive (TP), i.e., the number of malware samples are flagged as malicious by the ML model,

(2) False Negative (FN), i.e., the number of goodware samples are flagged as benign by the ML model,

(3) True Negative (TN), i.e., the number of malware samples are flagged as benign by the ML model, and

(4) False Positive (FP), i.e., the number of goodware samples that are flagged as malicious by the ML model.

Based on these enumerations, we can compute the following three metrics (cf. $Precison$, $Recall$, and $F1\ Score$) that are commonly taken as indicators for evaluating ML-based approaches, and are defined as follows.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall}$$

3.5 Experimental Results

In this section, we detail the experimental results we observed for answering our four research questions from Section 3.4.1. The four research questions cover both 10-fold cross-validation (RQ2) and holdout experiments (RQ 2-3), supervised learning (RQ 2-3) and unsupervised learning (RQ4) approaches, as well as a comparison of different machine learning algorithms (RQ3).

3.5.1 RQ2: Impact of Duplication Bias on Malware Classifiers

Our experiments to assess the impact of duplication bias in malware datasets follow the ML-based approach for malware detection proposed by Arp et al. [13] (i.e., we train the machine learning model through the famous SVM algorithm). We leverage the common datasets used in the Android malware detection literature (namely Drebin, AMD and Rmv-Droid) and investigate the impact of all the three levels of duplications (i.e., DEX file, Opcode Sequence, and API Call) in machine learning based malware detection. To better characterize the impact of duplication bias, following the general practice of state-of-the-art works [7, 130], we answer this research question in two experimental settings: (1) 10-fold cross-validation (also known as in-the-lab experiments), which is a widely used statistical method for estimating the capability of machine learning models. (2) in-the-wild experiments, which attempt to evaluate the capability of machine learning models in a real-world setting, i.e., training the model based on a known dataset and use the model to predict unknown dataset. Below we now describe these two experiments in detail.

3.5.1.1 10-fold Cross-validation

In 10-fold cross-validation, the dataset is randomly divided into 10 equal-size sample sets. Among the 10 subsets, one of them is retained as the test set for validating the performance of the machine learning model, which is then trained on the remaining 9 subsets. This process is then repeated 10 times with each of the subsets used exactly once as the test set. The performance measurements of these 10 validations are then averaged to compute the final performance metric of the classification approach.

Experimental Setup. Given a malware dataset and a duplicated sample type, we set up two experiments: one without sample duplication while another one with sample duplication. These two experiments form a controlled group for evaluating the difference brought by sample duplication to the machine learning-based classifications. To better distinguish the settings, we set up six experiments, two for each duplication type, for each malware dataset, as detailed below.

- **E1/E2: Without/With DEX Duplication.** These two settings evaluate the performance of machine learning approaches w.r.t. DEX duplication. For E1, i.e., without DEX duplication, the training set is formed by taking into account all the non-duplicated samples from the original dataset. For E2, we randomly select the same number of samples (as that of E1) from the original dataset to form the training set. Since the original dataset contains duplicated samples, the randomly selected subset will likely contain duplicated samples as well. Indeed, the chance of randomly selecting a set of $|E1|$ (i.e., 3,559) apps that is identical to E1 from the original dataset (i.e., 5,560 apps) is low.

- **E3/E4: Without/With Opcode Sequence Duplication.** Similar to **E1/E2** except that *opcode sequence duplication* is used instead of *dex file*.

- **E5/E6: Without/With API Call Duplication.** Similar to **E1/E2** except that *API call*

duplication is used instead of *dex file*.

Since we are interested in conducting binary classifications (i.e., malware or goodware) and two of the three considered datasets do not come with benign apps (the datasets contain malware only), we relied on the AndroZoo repository [9, 81, 94] to collect benign samples to train the machine learning model. AndroZoo is a growing collection of Android apps collected from several sources, including the official Google Play app market. It currently contains over 10 million Android APKs. Each of them has been scanned by over 70 anti-virus products hosted on VirusTotal[4]. We consider an app to be benign as long as none of the anti-virus products (hosted on VirusTotal) flags it as malware. Specifically, for each experimental setting, we randomly select the same number of goodware (as that of malware) to form the training set, as unbalanced training sets may introduce biases to machine learning based classifications. Furthermore, to avoid potential biases introduced by our random sampling, we ensure that there are no repackaged app pairs between the selected malware and goodware samples (i.e., do not share the same package names). We also conduct each experiment setting 10 times and report the average performance as the output. These settings apply to all the experimental results reported in this paper.

Result. Table 3.7 summarizes the 10-fold cross-validation results. Following the experimental setup, for each dataset, we perform six different experiments (i.e., E1 $\rightarrow$ E6): two experiments for a given sample duplication type. As indicated in the fourth column, different duplication types will result in a different number of samples for training, which subsequently leads to a different number of features (as shown in the fifth column). Generally, the more training samples considered, the larger the feature sets extracted.

The last six columns in Table 3.7 further illustrate the classification results for predicting both malware and goodware of the SVM-based malware detection approach. Based on these results, we can observe the following interesting findings:

[4]https://www.virustotal.com/gui/

Table 3.7: SVM-based Android malware detection via 10-fold cross validation. To support binary classification, we randomly select the same number of benign apps like that of malware to form the training set (i.e., #. malware *2).

Dataset	Type	Setting	Training Set	# Features	# Duplicated Vectors (Ratio)	Malware			Goodware		
						Precision(%)	Recall(%)	F1 score(%)	Precision(%)	Recall(%)	F1 score(%)
Drebin	Dex	E1 (Without Duplication)	3,559*2	48,026	949 (26.66%)	96.56	94.91	95.72	95.03	96.62	95.81
		E2 (With Duplication)	3,559*2	43711	2,019 (56.72%)	97.82	95.84	96.81	95.96	97.85	96.89
	Opcode seq	E3 (Without Duplication)	2,859*2	44,494	455 (15.91%)	96.2	94.58	95.38	94.71	96.28	95.48
		E4 (With Duplication)	2,859*2	40,305	1,338 (46.8%)	97.21	95.24	96.21	95.37	97.28	96.31
	API call	E5 (Without Duplication)	2,646*2	39,388	316 (11.94%)	96.18	93.91	95.02	94.11	96.28	95.17
		E6 (With Duplication)	2,646*2	35,117	1,162 (43.93%)	97.12	95.1	96.09	95.24	97.19	96.2
AMD	Dex	E1 (Without Duplication)	19,205*2	185,011	3,299 (17.18%)	98.05	95.64	96.78	95.87	98.04	96.9
		E2 (With Duplication)	19,205*2	172,507	4,730 (24.63%)	98.21	95.92	96.99	97.19	97.06	97.06
	Opcode seq	E3 (Without Duplication)	12,863*2	143,889	1,545 (12.01%)	97.78	95.77	96.71	96	97.79	96.84
		E4 (With Duplication)	12,863*2	122,438	4,629 (35.99%)	98.14	96.13	97.07	96.34	98.14	97.19
	API call	E5 (Without Duplication)	11,735*2	135,485	1,015 (8.65%)	97.48	94.93	96.09	95.32	97.5	96.32
		E6 (With Duplication)	11,735*2	114,330	3,895 (33.19%)	97.85	96.13	96.94	96.31	97.86	97.04
RmvDroid	Dex	E1 (Without Duplication)	8,893*2	100,998	1,398 (15.72%)	97.79	98.45	98.11	98.44	97.73	98.07
		E2 (With Duplication)	8,893*2	100,100	1,561 (17.55%)	97.81	98.49	98.14	98.48	97.75	98.11
	Opcode seq	E3 (Without Duplication)	6,122*2	77,693	432 (7.06%)	96.79	98.25	97.5	98.23	96.67	97.42
		E4 (With Duplication)	6,122*2	71,079	1,011 (16.51%)	97.15	98.4	97.75	97.77	97.72	97.72
	API call	E5 (Without Duplication)	5,453*2	73,467	273 (5.01%)	96.35	98.06	97.18	98.02	96.23	97.1
		E6 (With Duplication)	5,453*2	62,587	1,030 (18.89%)	97.3	98.33	97.89	98.32	97.23	97.75

Finding-2.1: When predicting goodware, no matter which metric is considered, or which duplication type is concerned, the performance achieved via training datasets containing duplicated samples is always higher than that of datasets without duplicated samples.

Finding-2.2: Regarding malware prediction, the results are more or less similar to that of goodware.

These two findings suggest that machine learning models tend to achieve higher performance if there are overlaps between the training sets and test sets. Indeed, when performing 10-fold cross-validations, duplicated samples will likely be divided into both training and test sets. This verdict further implies that machine learning approaches, in practice, should be trained on datasets that are as comprehensive and representative as possible. The more representative samples we can include in the training set, the more likely there will be similar samples in the test set, and thereby the higher the performance machine learning approaches could achieve. Nevertheless, we argue that researchers, when reporting the performance of their machine learning approaches, should make it clear if sample duplication exists in their dataset.

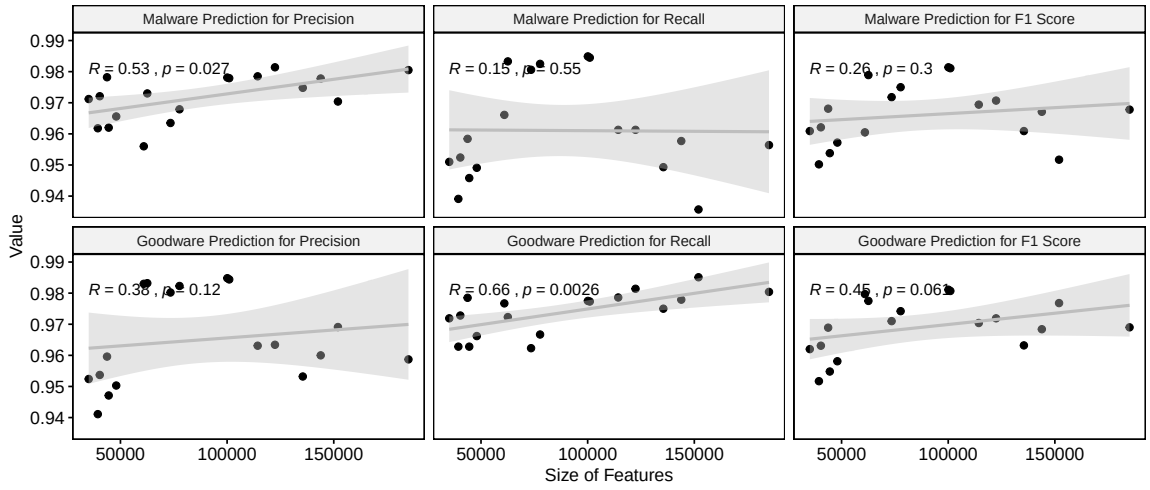


Figure 3.5: The correlation between feature size and the classification results (i.e., Precision, Recall, and F1 score). Each dot represents one experiment.

Interestingly, as demonstrated in Table 3.7, the performance difference between a controlled experimental pair E1 and E2, no matter which metric is concerned, is always smaller than 1.32%. This evidence suggests that the actual impact brought by sample duplication is insignificant. In other words, the validity of state-of-the-art malware detection approaches may not be severely threatened due to sample duplication of their training dataset. Nevertheless, we argue that duplicates (1) should still be removed to avoid unnecessary biases in machine learning-based classifications, or (2) be kept if a clear and convincing argument can be given.

Fig. 3.5 further illustrates the correlation between feature sizes and the performance of the machine learning approach. The fact that $R - value$ is positive for all the cases shows that the performance of the 10-fold cross-validation is generally aligned with the size of features considered. Nonetheless, some of the positive correlation is quite weak and yet not significant, given a significance level of $\alpha = 0.001$[5]. This finding further confirms that the impact brought by sample duplication might be marginal to the performance of machine

[5]There is one chance in a thousand that the difference between the datasets is due to a coincidence.

learning approaches.

3.5.1.2 In-the-wild Experiments

As advocated by Allix et al. [7], when conducting machine learning-based Android malware detection, in-the-lab experiments, such as using 10-fold cross-validation, may not be reliable to justify the performance of the machine learning models. There is also a need to validate the performance of the machine learning models in a real-world setting, i.e., in-the-wild experiments such as training on a dataset while testing on another dataset. In our second research question, we re-evaluate the impact of sample duplication for machine learning-based malware detection approaches through a so-called in-the-wild experimental setting.

Experimental Setup. As illustrated in Fig. 3.6, given a malware dataset, we create two subsets, namely ND and D. We put all samples one by one that do not introduce duplication into ND, and those that involve duplication into D. As a result, all the samples in ND do not contain duplicated samples while all the samples in D have duplicated versions presented in ND. We then randomly select a small set of samples, Y, in ND to form the test set[6] and prepare the training set in two settings, noted as S1 and S2 in Fig. 3.6).

In this experiment, we set Y as 30% of the samples in ND (i.e., 30% of samples with no duplication). Consequently, the training malware set contains $|70\%ND|$ samples (i.e., either $ND - Y$ or $35\%ND + X$, where X can be calculated via the following formula.

$$X = \begin{cases} D, & |D| <= |35\%ND| \\ K, & K \in D \; and \; |K| = |35\%ND| \end{cases}$$

In rare cases, if $|D| < 35\%|ND|$, meaning that we cannot select the same number of apps from the set of D to form the training set with duplication, we simply select all

[6]It is worth noting that the test set will have no impact on the performance of machine learning models. We kept the test set duplication-free to avoid potential biases.

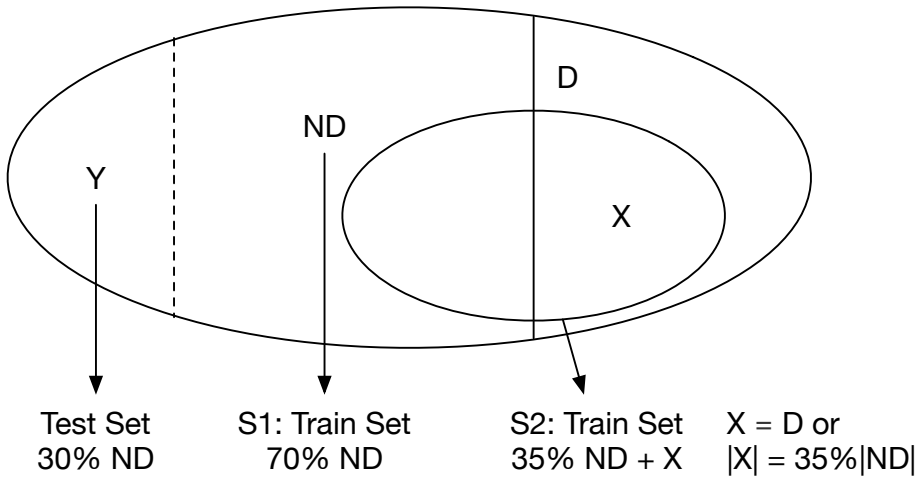


Figure 3.6: The experiment design of ML-based Malware Detection. X stands for the set of samples selected from $|D|$ while Y stands for the set of apps selected for fulfilling the test set, which is kept the same for both $S1$ and $S2$.

the apps in the D set to form the training malware set. Recall that we aim at conducting binary classification in this work. Therefore, we need to add goodware to the training set as well. To this end, we randomly select the same number of goodware (i.e., $|70\%ND|$) from Google Play to form the final training set.

Furthermore, instead of arbitrarily stipulating the values of hyper-parameters for the machine learning model, we leverage the *grid search* technique to automatically find suitable values for those hyper-parameters. *Grid search* is a popular pre-process step that explores all the possible parameter combinations to pinpoint an optimal combination for the model.

Result. Table 3.8 summarizes our experimental results for the experiments conducted for answering RQ2, for which SVM binary classification is applied to all the three representative datasets. When a different duplication type is involved, the total number of non-duplicated and duplicated samples will be different. In all datasets, Dex duplication yields the largest number of non-duplicated samples, followed by Opcode Sequence and API Call, respectively. The different size of non-duplicated samples subsequently causes

Table 3.8: Experimental results of SVM-based malware classification. $|ND|$ means the number of non-duplicated samples (after excluding duplicated ones). To enable binary classification, the train and test sets are fitted with randomly selected goodware.

Dataset	Type	Setting	$\|ND\|$	Training Set	Test Set	# Features	# Duplicated Vectors (Ratio)	Malware			Goodware		
								Precision(%)	Recall(%)	F1 score(%)	Precision(%)	Recall(%)	F1 score(%)
Drebin	Dex	E1 (Without Duplication)	3,559	2,491*2	1,068*2	34,799	581 (23.32%)	93.4	96.81	95.08	96.69	93.16	94.89
		E2 (With Duplication)	3,559	2,491*2	1,068*2	31,299	1,341 (53.85%)	94.34	95.31	94.82	95.27	94.27	94.77
	Opcode seq	E3 (Without Duplication)	2,859	2,001*2	858*2	33,398	255 (12.74%)	91.37	96.38	93.81	96.17	90.9	93.46
		E4 (With Duplication)	2,859	2,001*2	858*2	29,843	864 (43.17%)	92.92	94.54	93.73	94.44	92.8	93.61
	API call	E5 (Without Duplication)	2,646	1,852*2	794*2	30,289	206 (11.12%)	91.89	95.71	93.76	95.53	91.55	93.5
		E6 (With Duplication)	2,646	1,852*2	794*2	26,942	756 (40.81%)	92.76	94.74	93.74	94.63	92.6	93.6
AMD	Dex	E1 (Without Duplication)	19,205	13,444*2	5,761*2	135,015	2,745 (20.42%)	97.6	99.55	98.56	99.54	97.55	98.54
		E2 (With Duplication)	19,205	13,444*2	5,761*2	111,349	5,838 (43.42%)	97.91	99.33	98.61	99.32	97.88	98.59
	Opcode seq	E3 (Without Duplication)	12,863	9,004*2	3,859*2	107,270	909 (10.1%)	97.09	99.43	98.25	99.42	97.02	98.2
		E4 (With Duplication)	12,863	9,004*2	3,859*2	90,748	3,058 (33.96%)	97.2	99.17	98.17	99.16	97.13	98.13
	API call	E5 (Without Duplication)	11,735	8,215*2	3,520*2	101,418	585 (7.12%)	97.3	99.32	098.3	99.3	97.24	98.26
		E6 (With Duplication)	11,735	8,215*2	3,520*2	85,655	2,569 (31.27%)	97.2	99.41	98.29	99.39	97.14	98.25
RmvDroid	Dex	E1 (Without Duplication)	8,893	6,225*2	2,668*2	73,911	898 (14.43%)	96.85	98.99	97.9	98.96	96.78	97.86
		E2 (With Duplication)	8,893	3,353*2	2,668*2	73,211	1,066 (31.79%)	96.86	98.97	97.9	98.95	96.79	97.86
	Opcode seq	E3 (Without Duplication)	6,122	4,285*2	1,837*2	58,773	247 (5.76%)	95.1	99.4	97.2	99.37	94.88	97.07
		E4 (With Duplication)	6,122	4,285*2	1,837*2	50,276	869 (20.28%)	95.73	98.85	97.26	98.81	95.59	97.17
	API call	E5 (Without Duplication)	5,453	3,817*2	1,636*2	52,695	160 (4.19%)	95.85	98.84	97.32	98.8	95.72	97.24
		E6 (With Duplication)	5,453	3,817*2	1,636*2	44,554	680 (17.82%)	96.15	98.55	97.34	98.51	96.06	97.27

different training and test sets, which further yields different numbers of features for the machine learning classification. The number of features is dependent on the selected training dataset. Different apps may contribute to different features, although the same extraction strategy is applied.

- **Finding-2.3:** Unlike the results we obtained previously, i.e., via 10-fold cross-validation, in this experiment, as highlighted in the table, training set with sample duplications often achieves a higher precision yet lower recall for predicting malware than such settings without duplication involved in the training sets. This finding implies that, with duplication samples in the training set, the classifier is more conservative and less likely to flag a given app as malware, resulting in a lower false-positive rate and hence a higher false-negative rate. In opposite, without sample duplication, the classifiers are able to achieve better recalls. This could be explained by the fact that more diverse samples of malware[7] are learned by the classifiers, making them more knowledgeable to pinpoint unknown ones.

- **Finding-2.4:** Different datasets will yield different classification results, which are

[7]Given fixed number of malware, the more duplicated malware samples included, the less diverse and representative the malware set will be.

further aligned with the size of the training dataset, which is similar to the result obtained via 10-fold cross-validation. In our experiment, AMD has the largest training dataset and achieves the best performance in terms of distinguishing malware from benign ones. The impact of sample duplication on machine learning-based malware detection can vary from dataset to dataset. For example, the performance yielded by the AMD dataset is more stable than that of the Drebin dataset, which yields a larger range of vibration of results. This observation further suggests that the dataset quality is very important for machine learning based Android malware detection.

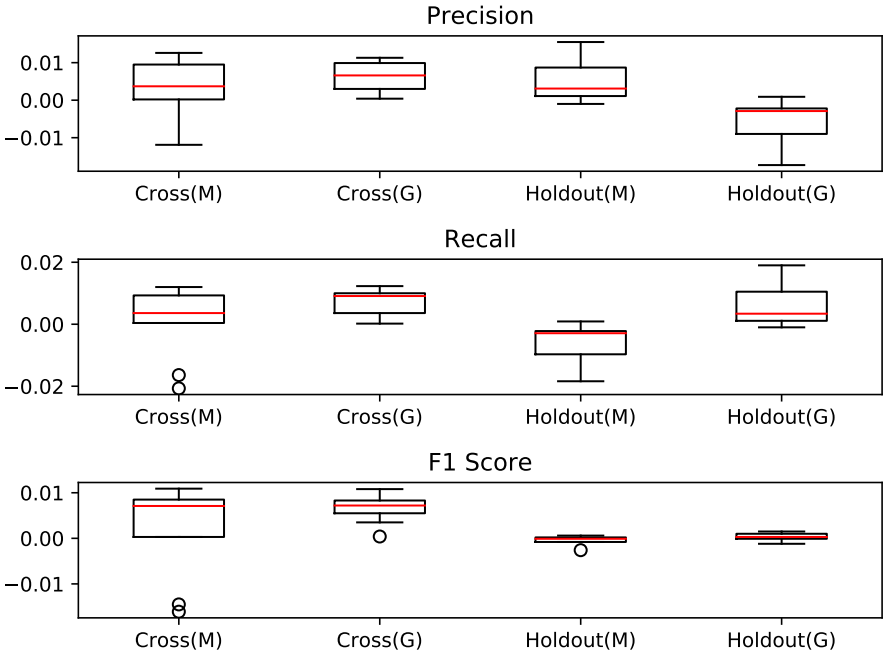


Figure 3.7: Distribution of performance differences between 10-fold cross validation and the holdout experiments.

RQ-2 Answer

When performing 10-fold cross-validation for Android malware detection, sample duplication has a positive impact on the performance of classification results. We hence advocate that practitioners and researchers should pay attention to sample duplication when conducting ML-based classification results. Nevertheless, the fact that the performance difference is quite small suggests that the impact brought by sample duplication to machine learning approaches might be marginal.
When performing in-the-wild experiments for predicting malware, sample duplication also impacts the performance of ML-based classification results. The fact that the performance of ML models varies from dataset to dataset further suggests that the dataset quality is important for ML-based Android malware detection approaches. Nonetheless, similar to that of 10-fold cross-validation, the impact of sample duplication on the machine learning models is marginal. We argue that duplicates (1) should still be removed to avoid unnecessary biases in machine learning-based classifications, or (2) be kept if a clear and convincing argument can be given.

3.5.2 RQ3: Impact of Duplication Bias on Different ML Algorithms

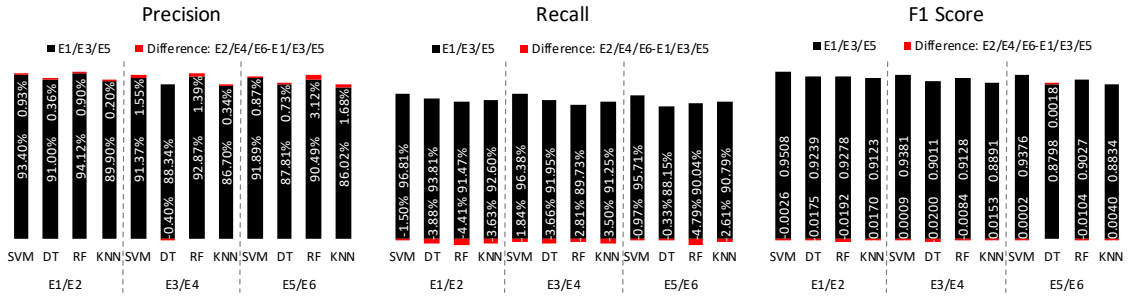


Figure 3.8: Experiment results with different ML algorithms on Drebin dataset.

As elaborated in Section 3.4.2, there are various machine learning algorithms recurrently leveraged by researchers for detecting Android malware. Specifically, we further consider three supervised machine learning algorithms, namely Decision Tree (DT), RandomForest (RF), and K-Nearest Neighbors (KNN).

Experimental Setup. The experimental setup for answering this research question is the same as the one leveraged for answering RQ2 (e.g., $E1 \rightarrow E6$) except that the machine learning algorithms are now altered to DT, RF, and KNN, respectively. Similar to the

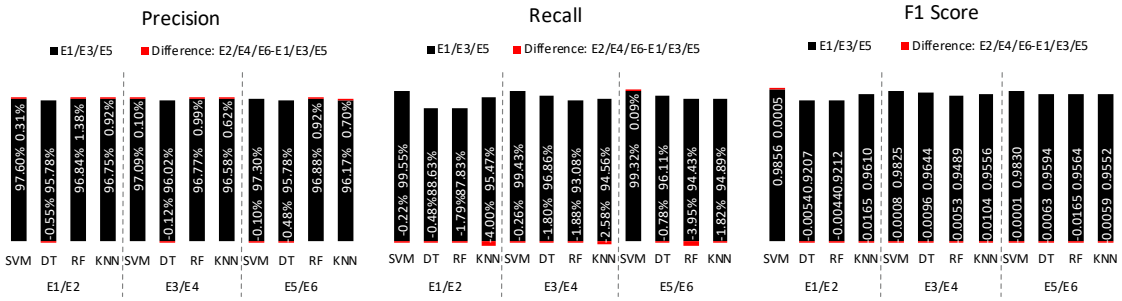


Figure 3.9: Experiment results with different ML algorithms on AMD dataset.

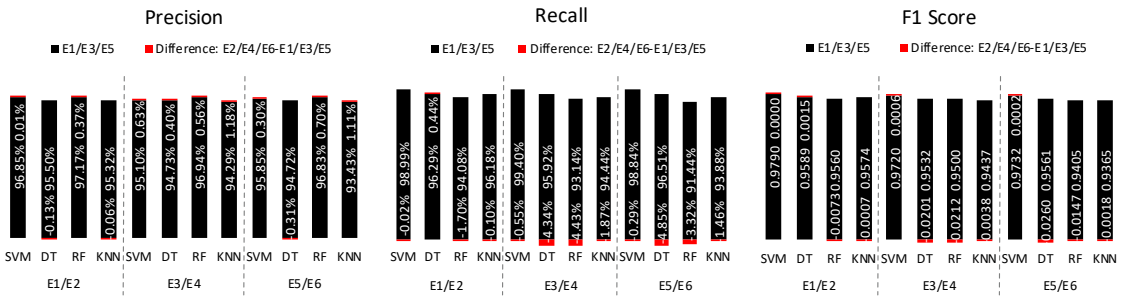


Figure 3.10: Experiment results with different ML algorithms on RmvDroid dataset.

experiments of SVM-based classification, we also leverage *grid search* to automatically stipulate hyper-parameter values for the newly selected machine learning algorithms.

Results. Fig. 3.8, Fig. 3.9, and Fig. 3.10 illustrate these experimental results. For each dataset, the precision, recall, and F1 score values are presented, and for each experimental setting, four machine learning algorithm results are comparatively illustrated. Based on these results, we summarize the following findings.

- **Finding-3.1:** When different evaluation metrics are considered, the performance of different machine learning algorithms is also different, although they are applied to the same dataset. Indeed, for precision, KNN yields the worst performance for Drebin and RmvDroid datasets, while DT yields the worst performance for the AMD dataset. For recall, RF yields the worst performance for all the three considered datasets. For F1 scores, SVM always achieves the best performance compared to the other ML algorithms.

- **Finding-3.2:** Sample duplication has diverse impacts on these different machine learning algorithms. As shown in Fig. 3.8-3.10, in the 36 controlled experimental groups – four machine learning algorithms are evaluated in each pair, e.g., SVM, DT, RF, and KNN in E1/E2, Precision of Drebin – DT appears to be impacted differently in 6 groups while SVM in 4 groups. There is no impact observed for RF and KNN.

Table 3.9 further summarizes the accuracy of the experiments. Accuracy is usually considered to be one of the most important metrics for checking if a machine learning classifier can be adopted in practice, as the cost of errors can be huge, e.g., requiring huge efforts for practitioners to review the results manually. Interestingly, as highlighted by the Δ column, which calculates the difference between two experimental settings (e.g., E2-E1), for almost all of the cases, the differences are positive. The machine learning classifiers trained based on a dataset without duplicated samples are generally more accurate than

Table 3.9: Accuracy of the classification with different experimental settings.

Dataset	Type	Setting	# Features	Δ	# Duplicated Vectors (Ratio)	Accuracy(%) SVM	Δ	DT	Δ	KNN	Δ	RF	Δ
Drebin	Dex	E1 (Without Duplication)	34,799	-3,500	581 (23.32%)	94.99	-0.2	92.27	-1.56	91.1	-1.51	92.88	-1.64
		E2 (With Duplication)	31,299		1,341 (53.85%)	94.79		90.71		89.59		91.24	
	Opcode seq	E3 (Without Duplication)	33,398	-3,555	255 (12.74%)	93.64	0.02	89.91	-1.83	88.62	-1.29	91.42	-0.61
		E4 (With Duplication)	29,843		864 (43.17%)	93.66		88.08		87.33		90.81	
	API call	E5 (Without Duplication)	30,289	-3,347	206 (11.12%)	93.63	0.04	87.96	0.24	88.02	-0.12	90.29	-0.73
		E6 (With Duplication)	26,942		756 (40.81%)	93.67		88.2		87.9		89.56	
AMD	Dex	E1 (Without Duplication)	135,015	-23,666	2,745 (20.42%)	98.55	0.03	92.36	-0.49	96.13	-1.49	92.48	-0.24
		E2 (With Duplication)	111,349		5,838 (43.42%)	98.58		91.87		94.64		92.24	
	Opcode seq	E3 (Without Duplication)	107,270	-16,522	909 (10.1%)	98.22	-0.06	96.42	-0.92	95.61	-0.94	94.98	-0.43
		E4 (With Duplication)	90,748		3,058 (33.96%)	98.16		95.5		94.67		94.55	
	API call	E5 (Without Duplication)	101,418	-15,763	585 (7.12%)	98.28	-0.01	95.94	-0.63	95.55	-0.52	95.7	-1.48
		E6 (With Duplication)	85,655		2,569 (31.27%)	98.27		95.31		95.03		94.22	
RmvDroid	Dex	E1 (Without Duplication)	73,911	-700	898 (14.43%)	97.88	0	95.88	0.13	95.73	-0.08	95.67	-0.65
		E2 (With Duplication)	73,211		1,066 (31.79%)	97.88		96.01		95.65		95.02	
	Opcode seq	E3 (Without Duplication)	58,773	-8,497	247 (5.76%)	97.14	0.08	95.29	-1.85	94.36	-0.27	95.1	-1.89
		E4 (With Duplication)	50,276		869 (20.28%)	97.22		93.44		94.09		93.21	
	API call	E5 (Without Duplication)	52,695	-8,141	160 (4.19%)	97.28	0.02	95.57	-2.45	93.64	-0.1	94.22	-1.28
		E6 (With Duplication)	44,554		680 (17.82%)	97.3		93.12		93.54		92.94	

such classifiers that are trained with datasets containing duplicated samples. A possible explanation for this effect could be that, compared to the latter case, the former setting contains more diverse malware samples, i.e., more malware characteristics, which could make the classifier more powerful in locating new samples. Nevertheless, the performance difference is still within a small range, no matter which machine learning algorithm is used.

RQ-3 Answer

Sample duplication can have diverse impacts on the performance of different machine learning models. However, no matter which machine learning algorithm is concerned, the impact seems to be marginal.

3.5.3 RQ4: Impact of Duplication Bias on Unsupervised Malware Clustering

In previous subsections, we have explored the impact of sample duplication on supervised learning approaches, w.r.t. three types of sample duplications that may have been overlooked by many state-of-the-art ML-based Android malware detection approaches. In this section we now explore the impact of sample duplication on unsupervised learning approaches, with a special focus on the duplication of malware families. The reason why unsupervised learning is selected is that it is one of the most common techniques used by

researches to identify Android malware families [29, 38].

Experimental Setup. Fig. 3.11 illustrates the process we followed to prepare the training sets for setting up the experiments in answering this last research question. The first step for setting up the experiments is to perform malware family analysis to identify the family of a given malware. Fortunately, all the datasets have been released with family labels associated with their samples. In this work, we directly leverage those labels to conduct our experiments.

Given a dataset, once the family labels are identified for all its malware, we rank the families based on the number of malware they are assigned to. For the sake of simplicity, and to better present the experimental results, we choose the top-10 families to form our experiments. For the malware of the top-10 families, given a duplication type, we separate the malware samples into two sets: ND (all the samples are non-duplicated from each other) and D (all the samples are duplicated to that of ND), following the same strategy we leveraged for setting up the experiments in answering RQ2. Based on this distribution, we form 2 experiments: one without duplicated samples (i.e., $S1'$, the ND set is directly leveraged) while another does contain duplicated samples (i.e., $S2'$ contains samples from both ND and D and the size is equal to that of $S1'$).

Finally, for these three types of duplication, we set up six experiments, two for each duplication type and form a control group.

- $E1'/E2'$: Two datasets respectively contain or do not contain DEX Duplication, i.e., $E1'$ utilizes $S1'$ and $E2'$ utilizes $S2'$.
- $E3'/E4'$: Similar to $E1'/E2'$ except that opcode sequence duplication is used instead of DEX Duplication.
- $E5'/E6'$: Similar to $E1'/E2'$ as well except that in this time API call duplication is used instead of DEX Duplication.

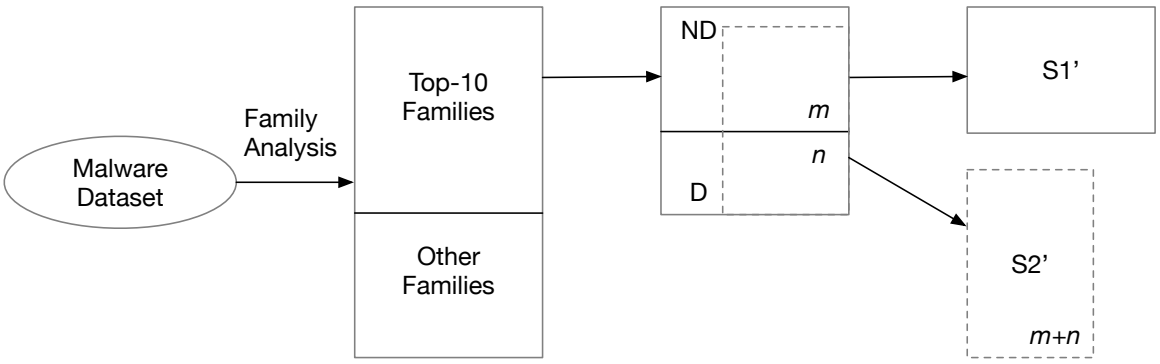


Figure 3.11: Training Sets Preparation for ML-based malware family classification.

Recall that our dataset for this experiment is formed by 10 families of apps. We hence set the final cluster numbers to 10, $k = 10$ for both K-means and Gaussian Mixture Model (GMM) [136] clustering algorithms[8], allowing for a better and clearer evaluation of the capability of the unsupervised learning models. In this setting, the learning model will group the input dataset into 10 clusters (e.g., clusters $1 \rightarrow 10$). Unfortunately, apart from grouping data samples into clusters, unsupervised learning approaches do not label the yielded clusters. To this end, after the clustering approach, we further leverage a straightforward approach to label the clusters. Specifically, given a cluster, we compare it to the inputted 10 family sets. We leverage the Jaccard similarity coefficient to calculate the distance between the given cluster and the original 10 malware family sets. Jaccard similarity coefficient is a simple yet well-known metric that has been frequently leveraged to calculate the similarity of sample sets, including the similarity of clusters categorized by unsupervised learning approaches [131, 95]. Given two clusters A, B, the Jaccard Index can be calculated as the ratio of the size of the intersection of A and B to the size of the union of A and B (cf. the formula below). The corresponding Jaccard index can be a value between 0 and 1, with 0 indicating no overlap (i.e., the two clusters are totally different) and 1 complete overlap

[8]Two of the most popular clustering algorithms. GMM can be regarded as an optimized version of the K-means model. In this work, we include two clustering algorithms to avoid potential biases.

(i.e., the two clusters are exactly the same) between the two clusters.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Results. Table 3.10 summarizes the experimental results of applying unsupervised learning approaches to cluster Android malware families, concerning Dex duplication in the training samples (i.e., $E1'$ and $E2'$). The ten selected malware families are shown in the first row of the table and the clustering results are enumerated in the second column, simply named as cluster $1 \rightarrow 10$. The value in each cell shows the Jaccard index between the samples in a cluster and the samples in a given family. $Distance = 0$ indicates that there is no overlap between the cluster and the given family.

For each cluster, we calculate its Jaccard indexes to all malware families and label it based on the family that achieves the largest index value. Let us take the second row as an example, for cluster 1 in $E1'$, we calculate 10 Jaccard indexes respectively for the 10 considered malware families, among which we obtain five positive indexes. Since the largest index goes to FakeInstaller, i.e., samples in this cluster are closer to the FakeInstaller family than others, we label this cluster as FakeInstaller.

To better present the difference between the two settings – with or without duplicated samples – we visualize the results in Fig. 3.12. Each ellipse represents a malware family. If the family is no longer identified after clustering, it will be highlighted with dotted lines. The families are connected through directed edges. Each directed edge represents a mis-clustering from source family to the target family. The mis-clustered sample numbers are further displayed as the weight of the edge, and which is also reflected by the thickness of the edge. For example, in the graph of $E1'$ in Fig. 3.12, there is an edge from *Kmin* to FakeInstaller. The weight $(14/85)$ indicates that there are 14 out of 85 Kmin malware being recognized as FakeInstaller malware. Fig. 3.13 further illustrates the visualization of mis-classified malware families for all the other experimental settings (for the results returned

Table 3.10: Jaccard distance between clustering results and the original family samples on Drebin.

Setting	Cluster No.	FakeInstaller	DroidKungFu	Plankton	Opfake	GinMaster	BaseBridge	Iconosys	Kmin	FakeDoc	Geinimi	Label
	Cluster 1	**0.356**	0	0	0.2218	0	0.0629	0.0031	0.0569	0	0	FakeInstaller
	Cluster 2	0.0045	0.1726	0.1048	0.0155	**0.2859**	0.0866	0.0148	0	0.0065	0.1058	GinMaster
	Cluster 3	0	**0.1917**	0	0	0.0115	0.0042	0	0	0	0	DroidKungFu
	Cluster 4	0	0	**0.7929**	0	0	0	0	0	0	0	Plankton
K-means E1'	**Cluster 5**	0	0	0	0	0	0	**0.625**	0	0	0	Iconosys
(Without Duplication)	**Cluster 6**	**0.2752**	0	0	0.2208	0	0	0.1569	0	0	0	FakeInstaller
	Cluster 7	0	0	0	0	**0.2644**	0	0	0	0	0	GinMaster
	Cluster 8	0	**0.4607**	0	0	0	0	0	0	0	0	DroidKungFu
	Cluster 9	0	0	0	0	0	0	0	0	**0.881**	0	FakeDoc
	Cluster 10	0	0	0	0	0	0.205	0	**0.5635**	0	0	Kmin
	Cluster 1	**0.5855**	0	0	0.1261	0	0	0	0	0	0	FakeInstaller
	Cluster 2	0.0327	**0.297**	0	0.0215	0.2336	0.0309	0.0036	0	0.0022	0	DroidKungFu
	Cluster 3	0	**0.5814**	0	0	0	0	0	0	0	0	DroidKungFu
	Cluster 4	0	0	**0.7127**	0	0	0	0	0	0	0	Plankton
K-means E2'	**Cluster 5**	0	0	0	0	0	0	**0.625**	0	0	0	Iconosys
(With Duplication)	**Cluster 6**	0	0.0133	**0.2429**	0	0.1939	0.0024	0	0	0	0.0027	Plankton
	Cluster 7	0	0	0	0	**0.2705**	0	0	0	0	0	GinMaster
	Cluster 8	0	0	0	0	0	**0.6822**	0	0	0	0	BaseBridge
	Cluster 9	0	0	0	0	0	0.085	0	0	0.2929	**0.55**	Geinimi
	Cluster 10	0.0859	0	0	0.2628	0	0.0079	0.1571	**0.336**	0	0	Kmin
	Cluster 1	0.0064	0.0934	0.0622	0.0586	**0.5248**	0.0561	0.0091	0.0172	0.0091	0.0174	GinMaster
	Cluster 2	0	0	**0.6902**	0	0	0	0	0	0	0	Plankton
	Cluster 3	**0.4676**	0	0	0.1501	0	0.0022	0.0968	0.2104	0	0	FakeInstaller
	Cluster 4	0	0	0	0.0035	0	0	**0.4658**	0	0	0.3646	Iconosys
GMM E1'	**Cluster 5**	0	**0.159**	0	0	0	0	0	0	0	0	DroidKungFu
(Without Duplication)	**Cluster 6**	0	**0.1405**	0	0	0	0.0049	0	0	0	0	DroidKungFu
	Cluster 7	0	0	0	0	0	**0.5273**	0	0	0.2791	0	BaseBridge
	Cluster 8	0	0	**0.1926**	0	0	0	0	0	0	0	Plankton
	Cluster 9	0	**0.5164**	0	0	0.0016	0	0	0	0	0	DroidKungFu
	Cluster 10	0	0	0	**0.1368**	0	0	0	0	0	0	Opfake
	Cluster 1	0	0.0655	0.0021	0.0045	**0.5714**	0.0407	‘ 0.0147	0	0.0054	0.0149	GinMaster
	Cluster 2	0	0	**0.3579**	0	0	0	0	0	0	0	Plankton
	Cluster 3	**0.4711**	0	0	0.082	0	0.0021	0.0734	0	0.1153	0.2051	FakeInstaller
	Cluster 4	0	0	0	0	0	0	**0.6316**	0	0	0	Iconosys
GMM E2'	**Cluster 5**	0	**0.3839**	0	0	0	0	0	0	0	0	DroidKungFu
(With Duplication)	**Cluster 6**	0	**0.5174**	0	0	0	0	0	0	0	0	DroidKungFu
	Cluster 7	0	0	0	0.0994	0	**0.449**	0	0.3761	0	0	BaseBridge
	Cluster 8	0	0	**0.6388**	0	0	0	0	0	0	0	Plankton
	Cluster 9	0	0	0	0	**0.2705**	0	0	0	0	0	GinMaster
	Cluster 10	0.0044	0	0	**0.3684**	0	0	0.0743	0	0	0	Opfake

by K-means only, the results returned by GMM are more or less the same and hence are not displayed to save space). From these visualized experimental results, Table 3.12 and Table 3.13 highlight the major differences returned by K-means and GMM, respectively.

Finally, we leverage MoJoFM [167], a Mojo distance based effectiveness metric, to measure the effectiveness of the selected two clustering models. The MoJoFM metric provides a more objective evaluation of the performance of clustering approaches, and can provide a single number as output that is simple to interpret and compare. Table 3.11 summarizes the experimental results. No matter which datasets or experimental settings are concerned, the differences between the MoJoFM scores achieved by K-means and GMM are not significant. However, no matter which clustering algorithms are concerned, there is a clear difference between a controlled pair of experimental settings (e.g., without or with duplicated samples).

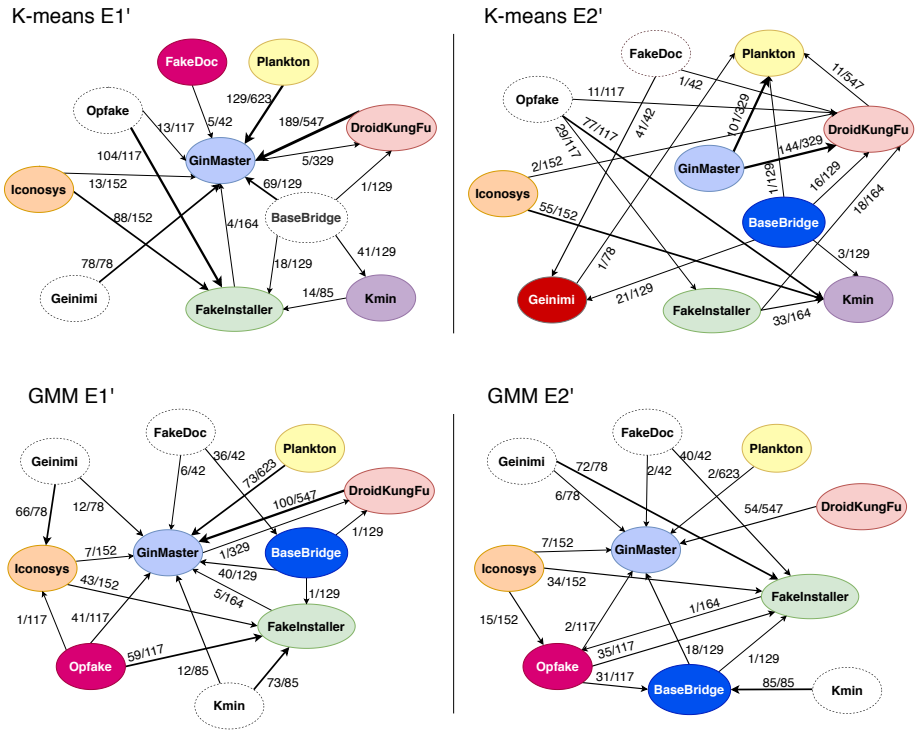


Figure 3.12: Visualization of the misclassified malware families (for $E1'/E2'$ of the Drebin dataset). The coloured shapes represent the retained families. The shapes with dotted lines represent such families that are not identified by the clustering approach.

Table 3.11: The MoJoFM Distance between the clustering results and the original partition.

Type	Setting	Drebin		AMD		RmvDroid	
		K-means(%)	GMM(%)	K-means(%)	GMM(%)	K-means(%)	GMM(%)
Dex	E1' (Without Duplication)	67.64	69.68	83.42	75.69	66.4	67.76
	E2' (With Duplication)	76.24	75.62	79.11	77.72	65.85	68.35
Opcode seq	E3' (Without Duplication)	70.07	76.35	81.26	81.75	57.72	61.81
	E4' (With Duplication)	73.76	78.34	75.02	74.55	58.38	58.83
API call	E5' (Without Duplication)	79.4	77.38	93.15	80.21	61.34	59.33
	E6' (With Duplication)	82.2	78.57	72.62	77.36	58.69	58.16

Finding-4.1: Sample duplication can indeed impact the performance of unsupervised learning models. For example, as shown in Fig. 3.12, given the same unsupervised learning model, e.g., K-means, the same malware dataset is clustered into 7 and 8 families (i.e., the colored shapes) for $E1'$ and $E2'$ (without and with dex duplication), respectively.

Finding-4.2: The unidentified malware families are almost always different between

the two experimental settings in a controlled pair. Indeed, as shown in the fifth column of Table 3.12 and Table 3.13, only one out of 9 pairs achieves the same result. This evidence suggests that, unlike that of supervised learning models, the impact of sample duplication on unsupervised learning models is quite significant.

Finding-4.3: The impact of sample duplication in unsupervised learning based malware classifications is independent of clustering algorithms. Likely, no matter which clustering algorithms are selected, the clustering results will be impacted by experimental settings without or with duplicated samples.

RQ4 Answer

Sample duplication has an impact on the performance of unsupervised machine learning models. The impact can be observed in all of our experiments with either different malware datasets or different duplication types. Furthermore, unlike supervised learning for which insignificant impact is observed, the impact of sample duplication on unsupervised learning is quite significant, and such an impact is independent of the underline selected learning algorithms.

3.6 Discussion

We now discuss the results of supervised learning with feature section (Subsection 3.6.1) and the experiment setting with realistic malware/goodware distribution in the test set (Subsection 3.6.2). We then stress the importance of considering sample duplication for machine learning (Subsection 3.6.3), and summarize the effect of parameter turning for ML-based malware detectors (Subsection 3.6.4) and the potential threats to validity of our study (Subsection 3.6.5).

3.6.1 Supervised Learning with Feature Selection

Recall that we have directly leveraged the features proposed by Arp et al. [13] to evaluate the impact of sample duplication in machine learning based Android malware detection. The set of features eventually considered in this work is hence comprehensive, which may

Table 3.12: Summary of major misclassified results with K-means. Since the number of samples selected from the Drebin dataset is smaller than the other two datasets, the number of errors obtained for the Drebin dataset is also fewer than that of the others.

Dataset	Type	Setting	#.Mistakes (Ratio)	Unidentified Families	Top 3 Error
Drebin	Dex	E1' (Without Duplication)	771 (34.02%)	BaseBridge, Geinimi, Opfake	DroidKungfu-189/547->GinMaster, Plankton-129/623->GinMaster, Opfake-104/117->FakeInstaller
		E2' (With Duplication)	565 (24.93%)	FakeDoc, Opfake	GinMaster-144/329->DroidKungFu, GinMaster-101/329->Plankton, Opfake-77/117->Kmin
	Opcode seq	E3' (Without Duplication)	538 (29.5%)	BaseBridge, FakeDoc, Geinimi, Iconosys, Opfake	Plankon-115/508->GinMaster, Iconosys-88/89->FakeInstaller, Opfake-81/81->FakeInstaller
		E4' (With Duplication)	408 (22.37%)	Iconosys, Kmin, Opfake	DroidKungFu-84/434->GinMaster, Plankton-71/508->GinMaster, Iconosys-57/89->Geinimi
	API call	E5' (Without Duplication)	295 (17.46%)	BaseBridge, Kmin, Opfake	Plankton-67/482->GinMaster, DroidKungFu-60/367->GinMaster, BaseBridge-43/74->GinMaster
		E6' (With Duplication)	297 (17.57%)	Kmin, Opfake	DroidKungFu-80/367->GinMaster, Plankton-60/482->GinMaster, Kmin-44/44->Geinimi
AMD	Dex	E1' (Without Duplication)	2,821 (17.05%)	BankBot, FakeInst, Fusob, Jisut	Youmi-796/1290->Dowgin, Airpush-500/7756->Dowgin, FakeInst-356/375->DroidKungFu
		E2' (With Duplication)	4,626 (27.95%)	BankBot, FakeInst, Fusob, Kuguo, Youmi	Airpush-1250/7756->Dowgin, Kuguo-1179/1180->Dowgin, Youmi-946/1290->Dowgin
	Opcode seq	E3' (Without Duplication)	1,953 (18.33%)	BankBot, Fusob, Jisut, Mecor, Youmi	Youmi-526/1079->Airpush, Youmi-461/1079->Dowgin, Airpush-228/4998->Dowgin
		E4' (With Duplication)	2,386 (22.4%)	BankBot, Fusob, Mecor, Youmi	Youmi-634/1079->Dowgin, Airpush-347/4998->Dowgin, Youmi-239/1079->Airpush
	API call	E5' (Without Duplication)	697 (7.24%)	BankBot, Fusob, Jisut, Mecor	Kuguo-124/890->Dowgin, Dowgin-89/2496->Airpush, Fusob-73/73->Dowgin
		E6' (With Duplication)	3,140 (32.61%)	BankBot, FakeInst, Fusob, Jisut, Kuguo, Mecor	Kuguo-886/890->Dowgin, Airpush-800/4470->Dowgin, Youmi-491/988->Dowgin
RmvDroid	Dex	E1' (Without Duplication)	2,646 (34.05%)	Dowgin, Gappusin, Kuguo, Viser	Airpush-510/2883->Adwo, Youmi-428/643->Adwo, Gappusin-426/456->Adwo
		E2' (With Duplication)	2,889 (37.18%)	Dowgin, Gappusin, Kuguo, Viser	Airpush-630/2883->Adwo, Youmi-430/643->Adwo, Gappusin-430/456->Adwo
	Opcode seq	E3' (Without Duplication)	2,180 (42.27%)	Dowgin, Gappusin, Mobidash, Viser	Airpush-366/1675->Kuguo, Gappusin-326/431->Kuguo, Youmi-307/570->Kuguo
		E4' (With Duplication)	1,924 (37.31%)	Dowgin, Mobidash, Viser	Youmi-329/570->Gappusin, Airpush-314/1675->Gappusin, Dowgin-249/272->Gappusin
	API call	E5' (Without Duplication)	1,738 (38.29%)	Dowgin, Mobidash	Airpush-275/1509->Gappusin, Youmi-253/506->Gappusin, Dowgin-194/267->Gappusin
		E6' (With Duplication)	1,879 (41.4%)	Dowgin, Gappusin, Mobidash, Viser	Airpush-302/1509->Kuguo, Gappusin-272/373->Kuguo, Dowgin-242/267->Kuguo

subsequently overfit the learning algorithm. Towards verifying this hypothesis, we replicate one of our previous experiments by integrating feature selection into the working process. Specifically, for the experiment conducted in Section 3.5.1.2, after the full feature set is extracted, we introduce a feature selection step into our approach aiming at retaining only such features that have importance weights higher than a given threshold. In this experi-

Table 3.13: Summary of major misclassified results with GMM.

Dataset	Type	Setting	#.Mistakes (Ratio)	Unidentified Families	Top 3 Error
Drebin	Dex	E1' (Without Duplication)	577 (25.46%)	Kmin, FakeDoc, Genimi	DroidKungfu-100/547->GinMaster, Kmin-73/85->FakeInstaller, Plankton-73/623->GinMaster
		E2' (With Duplication)	405 (17.87%)	Kmin, FakeDoc, Genimi	DroidKungFu-54/547->GinMaster, Kmin-85/85->BaseBridge, Genimi-72/78->FakeInstaller
	Opcode seq	E3' (Without Duplication)	386 (21.16%)	Opfake, BaseBridge, Iconosys, Kmin	GinMaster-64/328->DroidKungFu, Iconosys-65/89->GinMaster, Opfake-51/81->Genimi
		E4' (With Duplication)	517 (28.34%)	Opfake, Iconosys, Kmin, FakeDoc, Genimi	DroidKungFu-89/434->GinMaster, Geinimi-72/72->GinMaster, Plankton-52/508->GinMaster
	API call	E5' (Without Duplication)	411 (24.32%)	Opfake, BaseBridge, Kmin, Genimi	DroidKungFu-109/367->GinMaster, BaseBridge-67/74->GinMaster, Genimi-64/68->FakeInstaller
		E6' (With Duplication)	389 (23.02%)	Iconosys, Kmin, Genimi	GinMaster-91/328->DroidKungFu, Iconosys-74/75->Opfake, Genimi-66/68->Opfake
AMD	Dex	E1' (Without Duplication)	4,258 (25.73%)	FakeInst, Fusob, BankBot, Jisut, DroidKungFu	Dowgin-1265/3222->Kuguo, Youmi-632/1290->Kuguo, Airpush-536/7756->Kuguo
		E2' (With Duplication)	3,277 (19.8%)	Youmi, Fusob, BankBot, Jisut, DroidKungFu	Youmi-734/1290->Airpush, Dowgin-705/3219->Kuguo, Youmi-529/1288->Dowgin
	Opcode seq	E3' (Without Duplication)	3,465 (32.53%)	Mecor, Kuguo, BankBot, Jisut, DroidKungFu	Airpush-828/4998->Youmi, Kuguo-600/937->Dowgin, Dowgin-467/2617->Youmi
		E4' (With Duplication)	2,345 (22.01%)	Mecor, Fusob, Kuguo, BankBot, Jisut	Kuguo-691/937->Youmi, Airpush-460/4998->Youmi, Youmi-256/1079->Dowgin
	API call	E5' (Without Duplication)	1,969 (20.45%)	Mecor, Fusob, Kuguo, BankBot, Jisut	Kuguo-830/890->Dowgin, Airpush-381/4470->Dowgin, Youmi-201/988->Dowgin
		E6' (With Duplication)	2,487 (25.83%)	Mecor, Youmi, BankBot, Jisut	Youmi-524/988->Dowgin, Airpush-438/4470->Dowgin, Kuguo-255/890->Dowgin
RmvDroid	Dex	E1' (Without Duplication)	2,612 (33.62%)	Dowgin, Gappusin, Viser	Airpush-614/2883->Kuguo, Gappusin-345/456->Kuguo, Youmi-320/643->Kuguo,
		E2' (With Duplication)	2,423 (31.18%)	Dowgin, Gappusin, Kuguo	Kuguo-365/388->Airpush, Gappusin-309/456->Airpush, Airpush-277/2879->Viser
	Opcode seq	E3' (Without Duplication)	2,115 (41.01%)	Dowgin, Gappusin, Mobidash, Viser	Gappusin-302/431->Kuguo, Airpush-265/1675->Kuguo, Dowgin-247/272->Kuguo
		E4' (With Duplication)	2,169 (42.06%)	Dowgin, Mobidash, Mecor, Gappusin	Gappusin-285/431->Kuguo, Dowgin-241/272->Kuguo, Airpush-240/1675->Kuguo
	API call	E5' (Without Duplication)	1,870 (41.2%)	Dowgin, Mobidash	Kuguo-328/371->Airpush, Youmi-241/506->Gappusin, Dowgin-231/267->Airpush
		E6' (With Duplication)	2,173 (47.87%)	Dowgin, Gappusin, Adwo	Airpush-269/1507->Viser, Gappusin-263/373->Kuguo, Dowgin-240/267->Kuguo

ment, we set the threshold to be the average weights calculated based on the full feature set. Table 3.14 illustrates the new experimental results. The seventh column presents the numbers of selected features, which are significantly smaller (over 50%) than that of the original features (as shown in the sixth column). Nonetheless, by comparing with the experimental results shown in Table 3.8 (that obtained without involving feature selection),

Table 3.14: Experimental results (over the Drebin dataset) with feature selection applied. The features are selected only if their importance weights are higher than the average weights calculated based on the full feature set.

Type	Setting	$\|ND\|$	Training Set	Test Set	# Original Features	# Selected Features	Malware			Goodware		
							Precision(%)	Recall(%)	F1 score(%)	Precision(%)	Recall(%)	F1 score(%)
Dex	E1 (Without Duplication)	3,559	2,491*2	1,068*2	34,799	10,485	92.6	97.28	94.88	97.14	92.22	94.62
	E2 (With Duplication)	3,559	2,491*2	1,068*2	31,299	8,970	93.62	95.07	94.34	94.99	93.52	94.26
Opcode seq	E3 (Without Duplication)	2,859	2,001*2	858*2	33,398	10,024	90.5	96.73	93.51	96.49	89.85	93.05
	E4 (With Duplication)	2,859	2,001*2	858*2	29,843	8,789	92.23	94.78	93.48	94.63	92.01	93.3
API call	E5 (Without Duplication)	2,646	1,852*2	794*2	30,289	9,250	92.42	95.33	93.85	95.18	92.18	93.66
	E6 (With Duplication)	2,646	1,852*2	794*2	26,942	8,011	92.99	94.36	93.67	94.28	92.89	93.58

the experimental results are not significantly impacted by involving feature selection to the process. This evidence suggests that the selection of a large number of features has a limited impact on the experimental results of this work.

3.6.2 Realistic Malware/Goodware Distribution in Test Set

For all the experiments conducted in the evaluation section, we have followed many of the existing works by fulfilling the test datasets with balanced apps (i.e., containing the same number of malware and goodware). Unfortunately, this setting does not reflect the actual distribution of malware/goodware in the real world. Subsequently, the corresponding experimental results may not be able to represent the actual performance achievable in practice. We hence design additional experiments to check if such more realistic settings will impact our experimental findings. To the best of our knowledge, there is no ground truth about the actual distribution of malware/goodware, and it is non-trivial to obtain that in practice. Pendlebury et al. [130] have attempted to estimate such a ratio based on samples collected from the public AndroZoo dataset, which contains over 8 million apps at the time of their study. Eventually, they conclude that a reasonable estimation of malware to goodware distribution could be 1:9.

In this work, we take this distribution ratio to fulfil the additional experiments, i.e., by preparing new test/training datasets. For the sake of simplicity, since only the Drebin dataset has been provided with benign samples, we replicate the experiment (as presented

Table 3.15: Experimental results of SVM-based malware classification obtained based on realistic malware/-goodware distribution (i.e., 1:9).

Type	Setting	Training Set	Test Set	# Features	# Duplicated Vectors (Ratio)	Malware			Goodware		
						Precision(%)	Recall(%)	F1 score(%)	Precision(%)	Recall(%)	F1 score(%)
Balanced Training Set, Unbalanced Test Set											
Dex	E1 (Without Duplication)	2,491*2	1,068+9,612	34,799	581 (23.32%)	58.36	97.19	72.93	99.7	93.08	96.28
	E2 (With Duplication)	2,491*2	1,068+9,612	31,299	1,341 (53.85%)	63.66	95.06	76.25	99.48	94.58	96.97
Opcode seq	E3 (Without Duplication)	2,001*2	858+7,722	33,398	255 (12.74%)	60.75	96.62	74.59	99.54	92.16	95.71
	E4 (With Duplication)	2,001*2	858+7,722	29,843	864 (43.17%)	65.2	94.74	77.24	99.3	93.65	96.39
API call	E5 (Without Duplication)	1,852*2	794+7,146	30,289	206 (11.12%)	59.53	95.33	73.29	99.36	91.77	95.41
	E6 (With Duplication)	1,852*2	794+7,146	26,942	756 (40.81%)	62.68	94.31	75.31	99.23	92.87	95.94
Unbalanced Training Set, Unbalanced Test Set											
Dex	E1 (Without Duplication)	2,491+22,419	1,068+9,612	34,799	581 (23.32%)	90.82	91.75	91.28	99.18	99.07	99.13
	E2 (With Duplication)	2,491+22,419	1,068+9,612	31,299	1,341 (53.85%)	92.16	88.5	90.29	98.86	99.25	99.05
Opcode seq	E3 (Without Duplication)	2,001+18,009	858+7,722	33,398	255 (12.74%)	88.93	90.9	89.9	98.85	98.58	98.72
	E4 (With Duplication)	2,001+18,009	858+7,722	29,843	864 (43.17%)	90.87	86.76	88.76	98.34	98.91	98.63
API call	E5 (Without Duplication)	1,852+16,668	794+7,146	30,289	206 (11.12%)	90.9	89.41	90.15	98.66	98.86	98.76
	E6 (With Duplication)	1,852+16,668	794+7,146	26,942	756 (40.81%)	92.34	86.07	89.09	98.25	99.09	98.67

in Section 3.5.1.2) on the Drebin dataset only. The machine learning models are trained with the same algorithm and with the same dataset when a balanced training dataset is concerned or with the newly prepared unbalanced dataset. Table 3.15 summarizes the new experimental results. Interestingly, for the experiment of *Unbalanced Training Set, Unbalanced Test Set*, the experimental results are comparable to that achieved by *Balanced Training Set, Balanced Test Set*. When a balanced training set is concerned, i.e., *Balanced Training Set, Unbalanced Test Set*, while retaining very high recall of detecting malware as such, the precision has significantly decreased compared to the experimental results achieved by an unbalanced training set or a balanced test set. Nevertheless, similar to the findings we summarized previously, based on these new experimental results, differences can still be observed between such experiments trained with or without duplicated samples. This result once again suggests that sample duplication should be carefully considered (and avoided) when performing machine learning based Android malware detection.

3.6.3 The importance of sample duplication for machine learning.

In this work, we experimentally show that sample duplication indeed impacts the performance of machine learning-based Android malware detection approaches, w.r.t. both supervised and unsupervised learning models. This result aligns with the results reported

by Miltiadis Allamanis in investigating the adverse effects of code duplication in machine learning models of code [6]. In this work, we would like to emphasize that the impact of duplication on Android malware detection is quite marginal for supervised ML approaches. Unfortunately, the rationale behind this marginal impact is unclear at the moment. In our future work, we plan to fill this gap by conducting advanced explainable machine learning techniques.

Nonetheless, we argue that sample duplication could introduce biases depending on the ML-based classification approaches that may be used. We hence advocate that practitioners and researchers should pay more attention to sample duplication in their ML-based classifications. Ideally, sample duplications could be taken as a machine learning parameter, which needs to be explicitly communicated when reporting the performance of given machine learning approaches. Indeed, just like any other parameters of ML algorithms, such as k for the K-means algorithm, sample duplication rate is essential for supporting the reproducibility of the ML approaches. To help practitioners and researchers better communicate the sample duplications in their datasets for that of Android-oriented approaches, we further present to the community a prototype tool for characterizing duplicated samples in an Android app dataset. This tool further provides options for users to exclude duplicated samples from their datasets. We have made our prototype tool available online at `https://github.com/carol233/duplication`.

3.6.4 The effect of parameter turning for ML-based malware detectors

As empirically revealed by Allix et al. [7] in their large-scale empirical assessment of machine learning based malware detectors, no matter in which settings – 10-fold cross-validation or training on one set and test another set – RandomForest always achieves the best precision compared with other machine learning algorithms (including C4.5, RIPPER, SVM [7]). This empirical finding, surprisingly, is different from the one that we

observed in this work. We hence go one step deeper to check the possible reasons behind this difference. We followed the “Drebin” approach to set up our experiments, for which an additional “grid-search” step is adopted by searching for suitable parameter values for our learning algorithms. This is in contrast to the approach of Allix et al., who simply used the default options. Therefore, in this work, we re-launch all the experiments with the “grid-search” feature disabled. In this circumstance, RandomForest indeed jumps up to be the best learning algorithm for predicting Android malware. This contradictory result suggests that it is vital to tune ML algorithm parameter values when performing machine learning based malware classification. The algorithm that works best out of the box in default mode may not be the most suitable one if parameter turning is concerned [28, 47, 150]. This implication is further backed up by the fact that SVM rather than RaondomForest is adopted by the “Drebin" approach, although the RandomForest algorithm is frequently reported as one of the best algorithms in the literature.

3.6.5 Threats to Validity

The main threat to *construct validity* of our study concerns the exhaustiveness of classification algorithms we selected and the experiments we set up in this work. Although we have selected four well-known algorithms and both in-the-lab and in-the-wild experimental settings, which have also been frequently leveraged by other researchers to achieve similar purposes, they may not be entirely suitable for predicting Android malware [151]. Nonetheless, the four algorithms yield more or less similar results suggest that our findings are not specific to a particular learning algorithm. Another threat to *construct validity* lies in the process of preparing training/test datasets [150]. In this work, to ensure a balance between the size of training and test datasets, we choose a threshold of 30% to form the test set. This threshold may not be representative of this study. Ideally, to be fully conclusive, we would need to experiment with more thresholds. However, this is not the main focus

of this work, we leave it as future work. Furthermore, when preparing the training and testing datasets for evaluating the impact of sample duplication for supervised learning approaches, as shown in Fig. 3.6, there might be chances that some samples in the testing set have their duplicated counterparts set in the training set. This setting may lead to slightly higher classification performance as the malware detector could learn some malware information in advance. A more realistic setting would be to limit the testing samples to not include duplicated versions of the apps in the training set. An ideal approach could be to take app release time into consideration when preparing the training/testing set, e.g., testing apps are all released after the testing set, which is an ideal situation since the malware detector cannot learn from future samples, as suggested by Li et al [84]. Nevertheless, this is also not the main focus of this paper, we leave it as future work.

Yet another threat to *construct validity* concerns the feature extraction process of this work. Recall that, in this work, we directly leverage the feature set of the "Drebin" approach to train our machine learning models. However, the authors do not make their feature extraction scripts publicly available. To this end, we had to resort to the re-implementation of Annamalai Narayanan[9] to extract features from Android apps. Our re-implementation may not be identical to that of the original authors. Nonetheless, our re-implementation has been successfully adopted by both the authors themselves and many of our fellow researchers working in this community [117, 172]. Furthermore, the features extracted by the "Drebin" approach are mainly based on syntactic rules (e.g., the appearance of certain strings), which may not be able to characterize the semantic features of Android apps. Subsequently, the machine learning results might be impacted. In our future work, we plan to alleviate this impact by leveraging semantic features such as the ones extracted based on Android apps' graph representations [42] and advanced deep learning algorithms such as the ones driven by neural networks.

[9]https://github.com/MLDroid/drebin

The key threat to *internal validity* concerns possible errors in the implementation of our experimental tools and scripts used to run the experiments and gather experimental results. To reduce this threat, we have carefully reviewed the code and scripts of our toolchain to ensure that the implemented functions meet our expectations. We have further manually checked a random selection of experimental results to verify their accuracy.

One threat to the *external validity* of our study concerns the representativeness of the malware datasets that we selected. Although we have included four common malware datasets from the literature, our results may still not be generalisable to other malware datasets. Nonetheless, the fact that our experimental findings are similar among the selected datasets shows that the external validity of our work is likely to be reasonable. Also, to avoid potential biases, we restrict the test dataset to contain unduplicated samples only when conducting the supervised learning experiments, which unfortunately may not reflect the real-world situation as it is likely to have duplicated samples in a real-world dataset. Nevertheless, since this decision will not impact the capability of the classifier (which only relies on the training dataset) and the duplication rate in practice is not significant, such a decision should only bring limited threats to our experiments and hence could be neglected. In this work, the performance of our family clustering experiments is directly related to the authenticity of malware labels, which unfortunately may not be reliable, as often discussed by other researchers [65, 49, 147]. To mitigate this threat, we have directly leveraged the malware labels provided by the malware datasets, which have already been leveraged by various prior research projects.

3.7 Related Work

Machine learning-based Android malware classification has been a hot topic in the software engineering and security community. Below we summarize some representative prior work.

Android malware detection. Machine learning has been extensively leveraged by

practitioners and researchers to detect Android malware [128, 175]. One of the most common algorithms leveraged by researchers for achieving this purpose is RandomForest, which has been reported by researchers as one of the best algorithms for conducting binary classification. As an example, Alam et al. [5] have empirically demonstrated that RandomForest is optimal by comparing its accuracy with BayesNet, Logistic Regression, DT, etc. Later on, Allix et al. [7] have also empirically confirmed this. In their experiment, they experimentally show that RF achieves the best performance compared with C4.5, RIPPER, and SVM. A similar result has also been backed up by Li et al. [82] as well. In this work, although different datasets and feature sets are concerned, we achieve more or less similar results, i.e., RandomForest is among the best algorithm for precisely discriminating malware from goodware.

As discussed in the previous section, with "grid-search" enabled to optimise parameters, SVM in many cases can achieve an even better performance than that of RandomForest. Hence, SVM has also been a very common machine learning algorithm for training to predict Android malware. For example, Naser et al. [129] built a malware detector based on the features statically extracted from Android APKs. One of the most famous works that leverage SVM to predict Android malware is the one presented by Arp et al. [13]. They proposed the Drebin approach, for which they extract machine learning features from Android APKs (or DEX files) into eight feature sets. In our work, aiming at exploring the effect of sample duplication on machine learning based malware detectors, we leveraged the same feature sets and included SVM as one of our four evaluated machine learning algorithms. In many of our experimental settings, SVM indeed performs the best compared to that of other machine learning algorithms.

Most of the aforementioned approaches extract features statically from Android DEX files, which contain the core app code of the apps. In our work we have thus empirically explored the impact of DEX duplication on machine learning approaches. Apart from the

DEX file, we have also included two extra duplication types involving app opcode and API calls. These two types have been frequently leveraged by other researchers to form feature sets for learning the malicious behaviors of Android apps. Indeed, as an example, Jerome et al. [68] have proposed an ML-based malware detection approach based on opcode sequences in 2014. Similar to their work, Canfora et al. [27] and McLaughlin et al. [111] also respectively present machine learning based malware detection approaches based on features statically extracted from the raw Dalvik bytecode (i.e., opcode sequences). Since similar opcode sequences can be extracted from different apps, i.e., opcode sequence duplication, the performance of the approaches mentioned above might be impacted.

Similar to opcode sequences, Android APIs have also been recurrently leveraged as features for machine learning based Android malware analysis. For example, in 2013, Aafer et al. [3] have performed a thorough analysis that leverages critical API calls as features to evaluate the difference among selected classification algorithms. In their experiments, they employed four algorithms, including DT, C4.5, KNN, and linear SVM. Their experimental results reveal that KNN is the best algorithm for predicting malware when API calls are considered as features. This finding is quite different from ours as we experimentally show that RF and SVM are among the best algorithms. We note that our experiments are done on different datasets and use different feature sets, although API calls are considered by both approaches. Similarly, in 2016, Wu et al. [169] leveraged the use of dataflow-related API-level features to improve the performance of a KNN detector. We observe that approaches of leveraging API calls as features may be impacted by API call duplication if the authors do not carefully sanitize their training dataset.

In addition to traditional machine learning models, researchers have also started to leverage deep learning models to detect Android malware. In 2014, Yuan et al. [180] built a deep learning model with more than 200 features extracted from both static and dynamic analysis and stated that deep learning techniques are especially applicable for Android

malware detection. Likewise, in 2018, Karbab et al. [71] proposed an Android malware detection and family identification framework, MalDozer, which also leverages deep learning techniques to predict Android malware. In this work, we only focus on investigating the effect of sample duplication on traditional machine learning models. We nonetheless believe deep learning models are also relevant to the sample duplication concerns that we highlighted in this work. We plan to explore this direction in our future work.

Android malware family classification. In addition to machine learning-based malware detection, practitioners and researchers have also spent a significant amount of effort to identify the family of Android malware [52]. For example, Garcia et al. [52] proposed a novel approach for detecting malware families. By leveraging features extracted from specific Android API usages, reflective calls, and native binaries, they designed and implemented a prototype tool RevealDroid to achieve this purpose.

Most state-of-the-art approaches leverage unsupervised learning to identify Android malware families. The rationale behind this is that similar malware (belonging to the same family) will be grouped into the same cluster. As an example, Bayer et al. [19] have identified and grouped malware exhibiting similar behavior with a scalable clustering method. Similarly, in 2013, Hu et al. [61] designed and implemented a framework, namely MutantX-S, to cluster samples into families based on code instruction sequences efficiently. They have also proven that MutantX-S is highly accurate in detecting previously unknown malware. In 2015, Aresu et al. [12] created Android malware clusters by analyzing specific statistical information related to the HTTP traffic.

The above papers used clustering methods to aggregate malware with similar malicious behavior, which is of great significance for obtaining the family classification labels of malware. Unfortunately, none of these approaches has taken into account the sample duplication problem in their experimental setting, and thereby their performance might not be reliable.

Bias in machine learning. Apart from applying machine learning techniques to characterize Android malware, researchers have also started to investigate the potential biases that appear in the working processes of machine learning-based techniques. Pendlebury et al. [130] recently presented a study discussing the potential biases in two dimensions: space (referred to as spatial bias) and time (referred to as temporal bias). Spatial bias is caused by the unrealistic setting of the ratio of benign to malware samples in training and test data. Temporal bias refers to the integration of future knowledge about test data into the training stage. Similarly, Li et al. [84] have experimentally shown that time inconsistency introduces significant biases to machine learning based malware detection approaches.

In 2018, Li et al. [80] presented a study demonstrating that more features used by a machine learning approach do not necessarily mean better performance. In a recent work reported by Irolla et al. [66], 49.35% of the samples in the Drebin dataset have at least one more sample containing the same sequence of opcode. This result is in line with the findings of this work. Indeed it actually motivated us to investigate the potential impact of such duplication on the performance of machine learning approaches.

To the best of our knowledge, our work is the first to investigate the impact of sample duplication on machine learning-based Android malware detection approaches. However, studies on the adverse effects of code duplication in machine learning models have also been carried out. Allamanis et al. [6] presented a technical report describing the impact of multiple file-level (near-)clones appearing in large corpora of code. They discussed the biases introduced mathematically and empirically proved that code duplication can lead to overestimating the performance when evaluating machine learning models. Different from their work, our work in this paper targeting Android malware at the bytecode level.

3.8 Conclusion

In this paper, we empirically investigated the impact of sample duplication on machine learning-based Android malware detection approaches. We started by recognizing common sample duplication types in well known and used Android malware datasets. We then took into account these sample duplication types to train distinctive machine learning models to classify Android malware. We conducted our experiments on three common malware datasets. Our experimental results show that sample duplication does indeed impact the performance of machine learning-based malware detection approaches. An in-depth exploration further revealed that this finding applied to not only in-the-lab experiments (i.e., 10-fold cross-validation) but also in-the-wild analyses (i.e., trained on one dataset and then tested on another). This finding also applies to experiments that were conducted using different machine learning algorithms, including both supervised and unsupervised learning approaches.

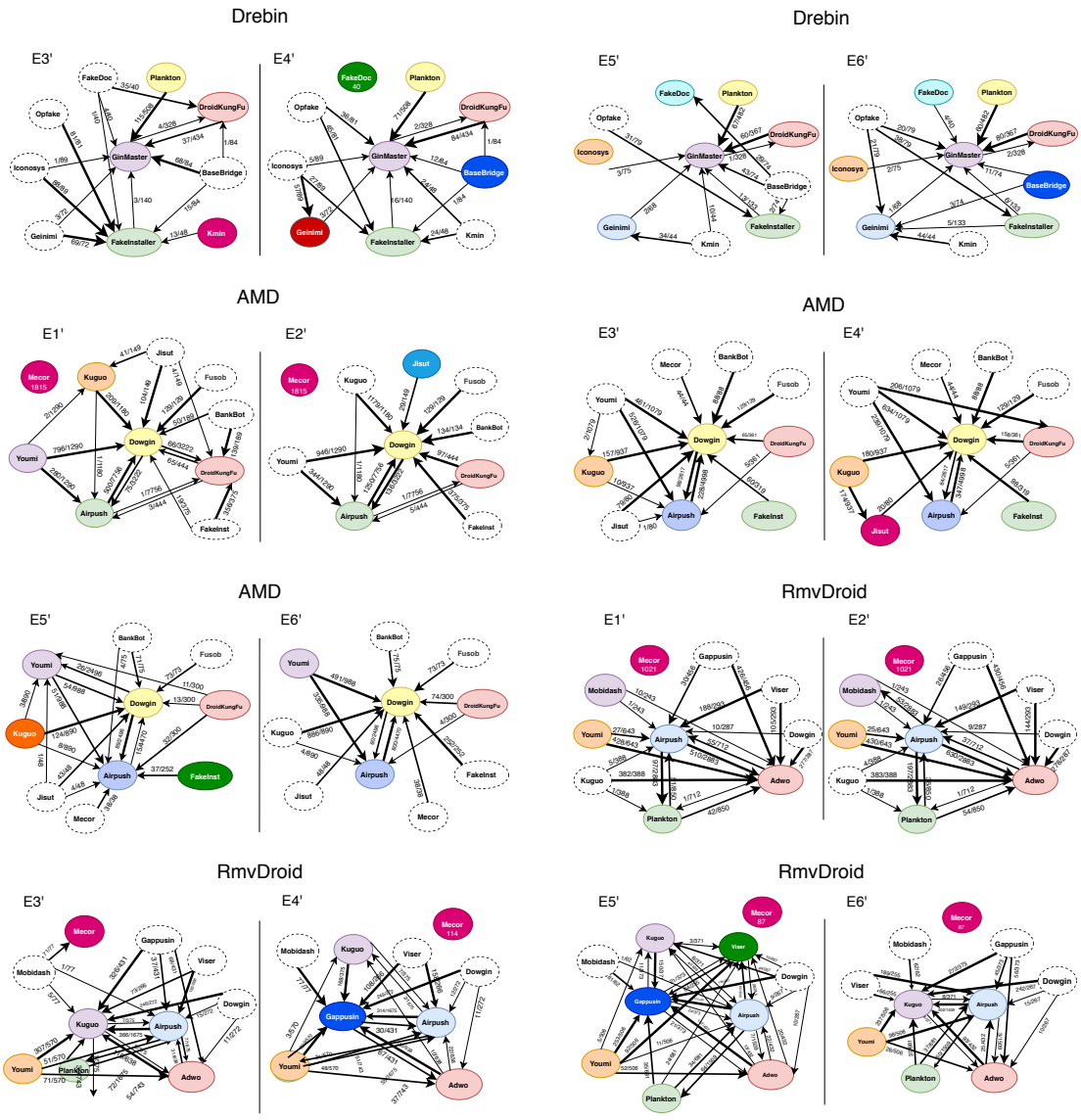


Figure 3.13: Visualization of the misclassified malware families (for all the other experimental settings).

Chapter 4

APIMatchmaker: Matching the Right APIs for Supporting the Development of Android Apps

Y. Zhao, L. Li, H. Wang, Q. He and J. Grundy, "APIMatchmaker: Matching the Right APIs for Supporting the Development of Android Apps," in IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2022.3146831.

4.1 Introduction

The software community has invented powerful IDEs (Integrated Development Environment) featuring comprehensive facilities, such as automatic code completion, to help developers better manage their software projects. The community has also made available a diversified set of libraries that offer Application Programming Interfaces (APIs) incorporating readily reusable function implementations. Developers can hence directly embed these libraries, instead of developing the same functions from scratch, to facilitate the development of their software applications.

Being able to provide readily reusable functions, APIs have become one of the most important components in modern software development. Our community hence has provided hundreds of thousands of software libraries, including APIs ranging from navigating maps, supporting security, using device features, and processing images and voice, etc. to scanning for malicious software packages. However, while the large number of existing

libraries provide convenience for experienced developers to implement software quickly, they also introduce significant burdens to developers. This is because they need to constantly spend significant time to learn the usage of each new APIs in detail to correctly deploy it. As argued by Robillard, some APIs are hard to learn, even for professional developers working at large software companies such as Microsoft [138]. Some have shown developers even spend up to 19% of their programming time on the internet to search for source code, especially API usage examples [21].

To mitigate this, much research effort has been spent to automatically recommend appropriate APIs and their usage patterns [124, 46, 122, 54, 72]. For example, Niu et al. [124] have proposed a clustering-based approach, which leverages the co-existence relations between object usages, to recommend API usages. Nguyen et al. [122] propose an approach leveraging predictive models such as Hidden Markov Model to recommend API usages. Gu et al. [54] and Kim et al. [72] propose to learn API usages from similar code examples through code search.

Unfortunately, most of the state-of-the-art works are either not targeted to Android developers, the main focus of our work, or are implemented based on techniques such as clustering or traditional predictive models that come with a number of drawbacks. Indeed, for the former case, existing works cannot be easily adapted to recommend APIs for Android developers because particular features need to be specifically fulfilled when developing Android apps [26, 91]. For example, since the Android framework evolves rapidly, it is non-trivial to develop Android apps supporting all the historically released framework versions, which nevertheless could still be used in outdated devices. To this end, developers usually develop apps targeting only a small range of Android frameworks (e.g., by specifying the minimal, targeted, and maximum SDK version the app is designed to support) rather than all the historical frameworks. As a result, when recommending APIs to Android app developers, the range of supported SDK versions need to be considered so

as to recommend usable APIs. For the latter case, clustering-based approaches, which use the frequency of patterns to achieve recommendations, has been demonstrated to be inefficient as *frequent patterns are often uninteresting patterns*[1], as demonstrated by Fowkes et al. [46]. Predictive model-based approaches often require users to manually label a training dataset and prepare features for learning, which are known to be time-intensive and prone to errors. Furthermore, as argued by Nguyen et al. [119], the existing pattern recommendation approaches also suffer from high redundancy and poor run-time performance.

To cope with these limitations, we propose a method for learning and recommending Android API usage patterns, based on concepts emerging from Collaborative-Filtering (CF) recommendation systems. These systems have been leveraged to recommend items for users to purchase. The recommendation is made by selecting items that have been bought by similar users in similar contexts. One core requirement to incorporate collaborative-filtering for recommendation is to assess the similarity of customers. As recommended by Nguyen et al. [119], the projects that are highly similar to the project under development should provide higher quality patterns than those of dissimilar ones.

In the context of recommending Android API usages, by considering the method intended to use them, i.e., the client code as "customers" and API methods as "products", the API recommendation problem can be reformulated as "*which API methods should the client code invoke in order to complete the method under editing, taking into account the fact that some APIs have already been invoked by the client code?*" Specifically, for the app under development, we would like to learn API usages from such apps that are most similar to it, so as to preferentially recommend APIs that are conjointly used by those similar apps.

Unlike existing approaches, which leverage code implementations to locate similar projects, in this work, we also consider app topics (which need to be provided by app developers), in addition to the pure code implementations. The rationale behind this is

[1]This problem is well known in the data mining literature [4].

that apps implementing the same topics tend to share similar high-level features, indicating likely similar (or even the same) code implementations. To this end, on top of the traditional collaborative-filtering algorithm, we propose a multi-dimensional context-aware collaborative-filtering approach to support the implementation of API recommenders for Android app development.

In this work, we were inspired by the FOCUS approach [119, 120], which was initially designed to learn from code snippets to recommend API usages for Java. The FOCUS authors have recently extended their initial implementation to further support the development of Android apps. They achieved this by directly converting Android apps code to Java code so as to be able to reuse the original Java-focused design. This indirect support has, however, overlooked some Android-specific features as they are not available in general Java applications. Our approach is designed to support the development of Android apps and hence has explicitly addressed those Android-specific features.

We design and implement a prototype API recommendation system called *APIMatchmaker* to support the development of Android apps. Since Android apps come with many different features compared to general Java applications, *APIMatchmaker* goes beyond the FOCUS approach by leveraging two-dimensional data (i.e., code implementation and app topic) to locate similar Android apps for learning and recommending API usages. We take into account both Android framework APIs and third-party library APIs. For the sake of simplicity, in this work, we will refer to both of them as Android APIs. Furthermore, due to the huge fragmentation issue (e.g., because of the Android framework's fast evolution), i.e., different Android apps may access different Android framework versions as each of them might contain a slightly different set of APIs, *APIMatchmaker* also takes this into account when recommending APIs.

With 12,000 real-world Android apps downloaded from AndroZoo [10] (their descriptions are directly crawled from Google Play), we experimentally demonstrate that our ap-

proach is effective and useful in recommending API usages to Android developers. *API-Matchmaker* achieves over 80% (or even 90%) of the success rate at Result@20 and can outperform the state-of-the-art approach as well as a baseline approach. The performance of *APIMatchmaker* can be even higher if the development of the active project is at a later stage, or increasing the size of the training app dataset, or by varying the customizable parameters provided by the tool. We make the following key contributions:

- we introduce to the community a new multi-dimensional context-aware collaborative filtering approach to better locate the most similar apps to support the recommendation;
- we design and implement a prototype tool called *APIMatchmaker*, which takes as input a method under editing and outputs a list of APIs (and their usage samples) meeting the constraints of the SDK versions that could be leveraged to complete the implementation of the method; and
- we evaluate our approach on 12,000 real-world Android apps under different experimental settings. Experimental results show that our approach is promising in recommending API usages to Android app developers.

The rest of this paper is organized as follows: Section 4.2 presents a motivating example attempting to help readers better understand the problem targeted in this work. Section 4.3 then details the design and implementation of our approach, namely *APIMatchmaker*. Next, we present the experimental setup and the evaluation results of our approach in Section 4.4 and Section 4.5, respectively. After that, we discuss the sensitivity of our approach concerning several aspects in Section 4.6. Finally, Section 4.7 discusses related work, and Section 4.8 concludes this paper.

```
1   private static void a(Context context, File file, String str, int i) {
2       if (context != null && file != null) {
3           String absolutePath = file.getAbsolutePath();
4           ContentValues contentValues = new ContentValues();
5           contentValues.put("_data", absolutePath);
6           ...
7           contentValues.put("_size", Long.valueOf(file.length()));
8           Uri contentUriForPath = MediaStore.Audio.Media.getContentUriForPath(absolutePath);
9           context.getContentResolver().delete(contentUriForPath, "_data=\"" + absolutePath + "\"", (String[]) null);
10          RingtoneManager.setActualDefaultRingtoneUri(context, i, context.getContentResolver().insert(contentUriForPath, contentValues));
11      }
12  }
```

Figure 4.1: Code snippet extracted from app *com.appsfreeinc.zebra.sounds* (A Zebra sound player).

4.2 Motivating Example

The typical usage scenario of our work is to recommend APIs, including their usages to app developers who are actively implementing an Android app. The developer might have already completed the development of some methods and is now halfway through finishing the method under editing (hereafter referred to as the "active method"). Fig. 4.1 illustrates such an example (extracted from app *com.appsfreeinc.zebra.sounds*, hereinafter referred to as the "active app project"). The active method (i.e., *a()*) is divided into two parts. The first part (lines 2-7) presents the code that has just been completed by the developer. The second part, highlighted in bold, is the actual implementation of this method (i.e., ground truth). In this motivating example, we consider the second part is unknown, and our objective of this work is to *recommend appropriate APIs for helping developers complete the second part*.

We plan to learn API usage patterns from existing apps because we hypothesize that similar apps may implement the same functionality using suitable, reusable libraries for the active app project. To this end, we use a large set of Android apps to locate similar apps of the active app project. Fig. 4.2 highlights two such example apps (B and C, respectively for apps *com.lahcenappsinc.drum.roll.sounds* and *com.andromo.dev137436.app236697*). Although both of them are very similar to the motivating example app in terms of code implementations (including the code in other methods that are not listed here), the API

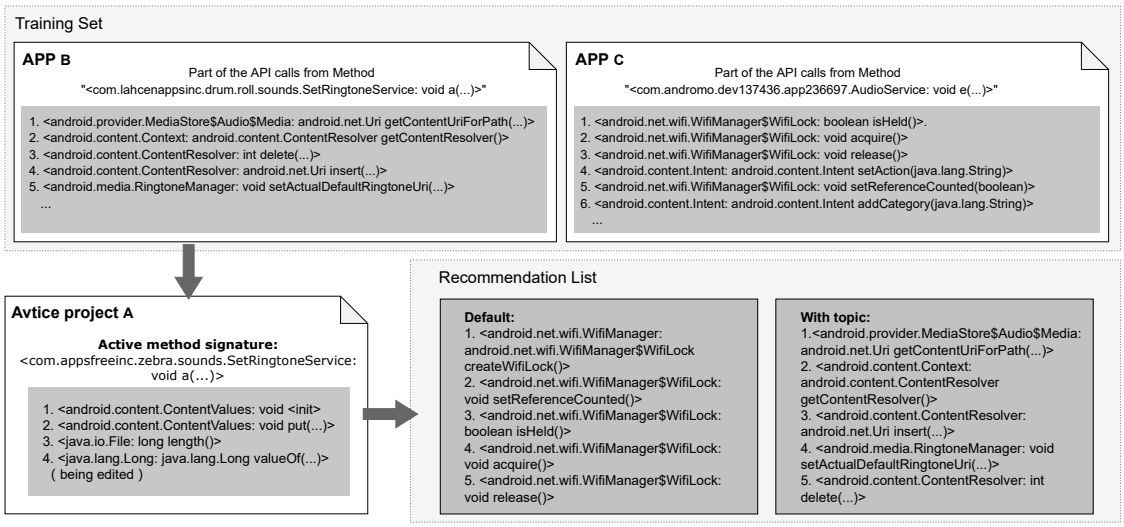


Figure 4.2: The details of active project A (under development), APP B, and APP C.

usages leveraged by these two apps are quite different, which could, in turn, provide noisy recommendation results.

Based on the recommendation algorithm proposed by Nguyen et al. [119] for recommending API usages for supporting the development of Java projects, for our motivating example, the recommended output would be the *default* list shown in Fig. 4.2. The result is however quite far from the ground truth highlighted in Fig. 4.1. We then look into the algorithm and find that in this example, app C has dominated the recommendation. Although both apps B and C are possible candidate apps to reuse API usage examples, a detailed analysis reveals that both app B and the active app project are players for offline sound resources while app C is for playing online music and news. Ideally, app B should be closer to the active project than app C and should contribute more to the recommendation. To this end, we refine our recommendation algorithm to also take into account the similarities of the app topic as codified in the app description text. The re-computed recommendation output, as highlighted in Fig. 4.2 (with topic), is now much more closer to our ground truth. This simple motivating example suggests that app topics should not be overlooked when

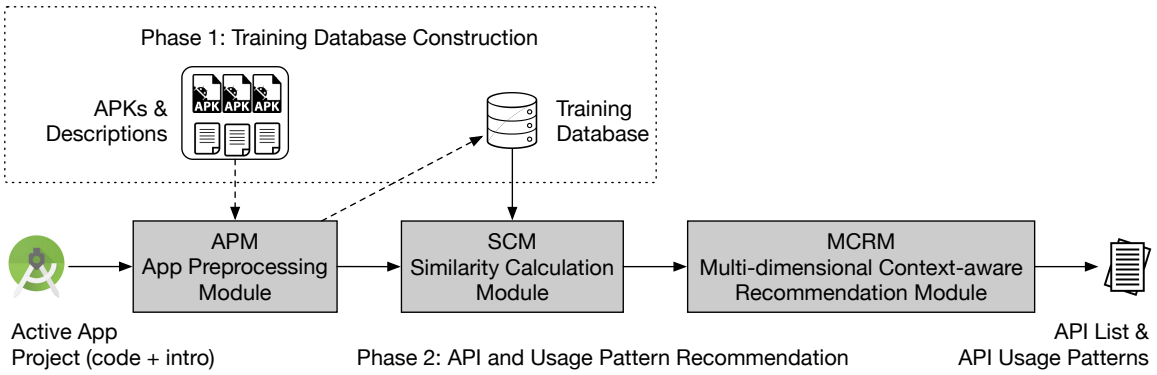


Figure 4.3: The architecture of *APIMatchmaker*.

learning implementations from existing Android apps. In this work, we leverage both other app implementations and other app topics to learn and recommend API usage patterns for the development of Android apps.

4.3 Our Approach: *APIMatchmaker*

Fig. 4.3 presents an overview of our approach *APIMatchmaker*, which leverages two phases to find the most likely best-fit existing Android APIs to assist in supporting the development of new Android apps. The two phases are (1) Training database construction and (2) API and usage pattern recommendation. The first phase prepares a training database based on the implementation of a large set of real-world Android apps. The training database will then be referenced and reused to support the implementation of the second phase, which aims to recommend APIs and API usage patterns to an active app project that is currently under development.

As shown in Fig. 4.3, *APIMatchmaker* is made up of three main modules: (1) App Pre-processing module (APM), (2) Similarity Calculation Module (SCM), and (3) Multi-dimensional Context-aware Recommendation Module (MCRM). The first module APM is leveraged by both of the two phases, while the remaining two modules are used to achieve

the objectives of phase 2.

4.3.1 APM: App Preprocessing Module

APM is used to process three types of inputs in order to serve both of the aforementioned phases: (1) Android APKs, (2) Active App projects under development, and (3) App descriptions. The first type is needed to fulfill the requirement of phase 1 towards constructing a training database for supporting API recommendation. Compiled Android APKs are leveraged rather than open-source code because we want to build our training database based on real-world – likely high-quality – Android apps, which are usually only released as APKs. The second type is the ideal input for the second phase, for which the overall goal of this work is to help developers complete their active app projects under development. The last input type is considered because similar apps – those implementing the same goal, embedding the same features, etc. – may be more likely to use APIs similar to the active app project and the active method, as stated by Nguyen et al. [119]. This information can be taken into account when learning API usages from Android apps. We leverage app descriptions to identify similar apps and this input type is needed to fulfill the purposes of both phases.

This leads us to three key data sources for our pre-processing module:

- **Android APKs** are closed-source app versions distributed over popular app markets such as the official Google Play store. This module leverages Soot to parse closed-source compiled app code at the Jimple code level to extract all the methods implemented in the app and the Android APIs accessed by those methods. In other words, our approach does not require converting Android bytecode to Java source code to achieve its purpose. Soot is an optimization framework for supporting static program analysis of Java and Android apps, and Jimple is a typed 3-address intermediate representation provided by Soot to ease its code manipulations [14]. The output

of this processing is the app's full list of method signatures and the set of Android APIs associated with each method signature.

- **Active App project under development.** The objective of *APIMatchmaker* is to match the right APIs for supporting the development of new Android apps. The input of *APIMatchmaker* is hence an app project that is under active development. In such a project, we expect that some methods have already been fully completed, while some others have not (might be half-completed or not even started). The goal of *APIMatchmaker* is hence to recommend the best suited APIs and their usage patterns that will help app developers quickly expand those incomplete methods.

  Since the active app projects come with source code, the pre-processing of the first module (APM) is achieved by directly parsing the source code. In practice, *API-Matchmaker* leverages JavaParser to ease the implementation. Given an active app project, this module will parse all its developer-defined methods and output those methods, along with their accessed Android APIs.

- **App descriptions** are provided by app developers to introduce and advertise the app. For closed-source apps, their descriptions can usually be collected from the app markets where the apps are hosted. For example, on the official Google Play store, all apps are provided with a dedicated page to describe their goals, functions and key features. For such apps that are still under development, we need their developers to provide such descriptions when using our tool-chain to help them implement the apps.

  App descriptions are provided in natural language. This module hence leverages natural language processing (NLP) techniques to pre-process this type of input data. Specifically, *APIMatchmaker* will first cut the description text paragraphs into words. It then turns all the words into lower case, and removes punctuation and stop words

(i.e., the most common words in a language). After that, *APIMatchmaker* performs a stemming step to further remove the morphological affixes of the remaining words to only retain their common base forms. Due to grammatical reasons, different forms of a word, such as *organize*, *organizes*, and *organizing*, could be used. However, these forms are essentially just different tenses of the same root word, i.e., *organize*, and hence should be treated as such. The last stemming step is introduced to achieve that, i.e., mapping words that are derived from one another to a common word.

4.3.2 SCM: Similarity Calculation Module

Given an Android app project under active development, our *APIMatchmaker*'s second module performs similarity analyses aiming to find similar apps from the training set. As shown by Zhong et al. [187], under certain usage scenarios (e.g., implementing the same functions or leveraging the same library modules), API methods are frequently called together and even follow some sequential calling rules. Taking this empirical evidence in mind, we hypothesize that similar apps under specific scenarios could share similar API usage patterns as well. Therefore, we believe that it is also possible to learn API usage patterns from existing apps.

To this end, in this Similarity Calculation module, we aim to identify and learn API usages from existing apps that are similar to the app projects under active development. As shown in Fig. 4.4, we rely on two steps to achieve this purpose. First, we leverage code implementation and app description to select x apps[2] that are most similar to the app project under development. Then, for such methods under editing in the active app project, we select top-y similar methods from the previously selected x apps. These y methods will be leveraged by the MCRM module (detailed in Subsection 4.3.3) to learn and recommend API usages for the methods under editing in the active app project.

[2]This parameter is configurable. Similarly, the upcoming parameter y is also configurable.

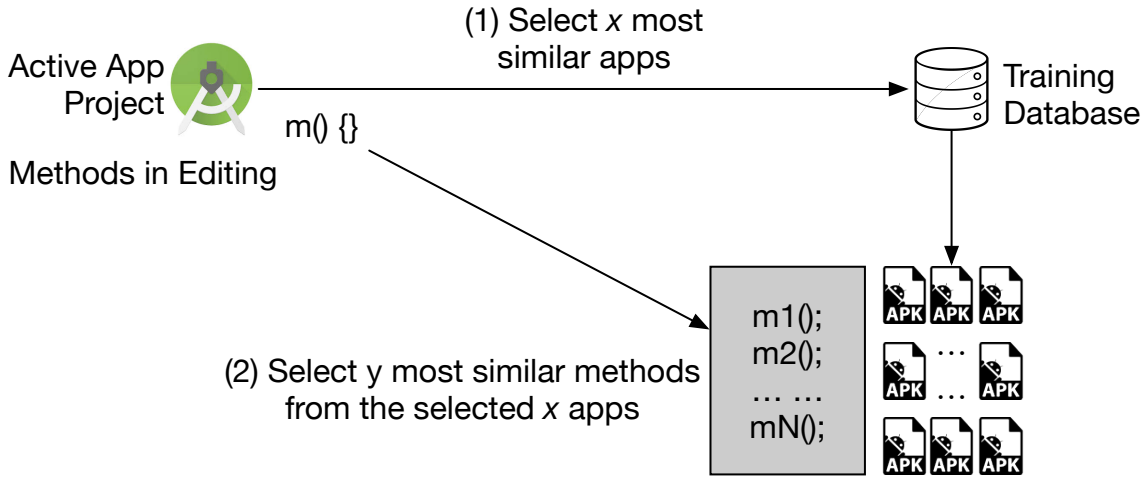


Figure 4.4: The process of selecting x most similar apps and y most similar methods (x and y are parameters needed to be customized by the users of *APIMatchmaker*).

4.3.2.1 Select x most similar apps.

For this we use the following two types of similar apps: (1) apps that are similar in terms of their targeted topics or features. The rationale behind this is that apps implementing the same topics could share similar code implementation, such as leveraging the same third-party libraries. (2) apps that are similar in terms of app implementation. These apps may share the same method signatures or access the same set of Android APIs compared with the app project under active development.

App Similarity (Topic). For calculating the topic similarity of Android apps, app descriptions are leveraged to fulfill this purpose. Using the first pre-processing module, the app descriptions have been transformed to clean and concise versions. We leverage a popular and classical algorithm called Term Frequency and Inverse Document Frequency (TF-IDF) to calculate the topic similarities between the active app project and the apps in the training database.

In the TF-IDF algorithm, Term Frequency (TF) refers to the frequency of keywords in a document, which can be calculated via the following formula, where $n_{i,j}$ is the number

of times the word i appears in document j, and $\Sigma_k n_{k,j}$ is the total number of words in document j.

$$TF_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{4.1}$$

Inverse Document Frequency (IDF) refers to the inverse text frequency, which is an index used to measure the weight of keywords. IDF can be calculated by the following formula, where $|D|$ is the total number of documents, and $|j : t_i \in d_j|$ is the number of documents where the word i presents.

$$IDF_i = log\frac{|D|}{|j : t_i \in d_j|} \tag{4.2}$$

Generally speaking, the higher frequency of words in a particular document (i.e., higher TF value), or the lower frequency of words in the entire document set (i.e., lower IDF value), the higher TF-IDF value can be achieved. In other words, TF-IDF tends to filter out common words, meanwhile retaining the important ones. The TF-IDF can be calculated via the following formula:

$$TF - IDF = TF_{i,j} \times IDF_i \tag{4.3}$$

Given two app descriptions p' and q' (of apps p and q) and their TF-IDF vectors $\vec{\lambda}$ and $\vec{\mu}$, respectively, *APIMatchmaker* leverages cosine similarity to calculate the distance of these two descriptions (cf. Formula 4.4).

$$sim_1(p, q) = \frac{\vec{\lambda} \cdot \vec{\mu}}{|\vec{\lambda}||\vec{\mu}|} = \frac{\sum_{i=1}^{n} \lambda_i \times \mu_i}{\sqrt{\sum_{i=1}^{n} (\lambda_i)^2} \times \sqrt{\sum_{i=1}^{n} (\mu_i)^2}} \tag{4.4}$$

App Similarity (Implementation). Similar to the approach used to calculate the topic similarity of Android apps, we leverage the same TF-IDF algorithm to calculate app similarities based code implementations. The only difference is that API calls are leveraged

rather than descriptive natural language words. We represent an Android app as a vector $\vec{\phi} = \{\phi_i\}_{i=1,...,k}$, with ϕ_i being the TF-IDF value of each API call. Then, the similarity of apps p and q, with their feature vectors $\vec{\phi} = \{\phi_i\}_{i=1,...,k}$ and $\vec{\omega} = \{\omega_j\}_{j=1,...,l}$, can be calculated by Formula 4.5.

$$sim_2(p,q) = \frac{\vec{\phi} \cdot \vec{\omega}}{|\vec{\phi}||\vec{\omega}|} = \frac{\sum_{t=1}^{n} \phi_t \times \omega_t}{\sqrt{\sum_{t=1}^{n}(\phi_t)^2} \times \sqrt{\sum_{t=1}^{n}(\omega_t)^2}} \tag{4.5}$$

4.3.2.2 Select y most similar methods.

The reason why this step is conducted rather than simply taking into account all the methods declared in the x apps is that many of the declared methods may not access or only access a few Android APIs. Hence, it could introduce noise into our approach if those methods are taken into account. Therefore, in this work, we decide to only select y most similar methods to support further analyses.

Method Similarity. Given the method under editing for which we are about to recommend APIs and usage patterns to use, we leverage the Jaccard similarity coefficient to find its y nearest neighbors $M = \{m_i\}_{i=1,2,3,...,y}$. The Jaccard coefficient is a well-used metric that has been frequently leveraged to calculate the similarity or distance of different sets. Given two sets A, B, as shown in Formula 4.6, the *Jaccard Index* is expressed as the ratio of the size of the intersection of A and B to the size of the union of A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.6}$$

For our work, we calculate the similarity between the method under editing and a method in the x apps via Formula 4.7, where $\mathbb{F}(m)$ and $\mathbb{F}(n)$ are the sets of API calls extracted from method signatures m and n (the extraction is done in the APM module).

$$sim_\gamma(m,n) = \frac{|\mathbb{F}(m) \cap \mathbb{F}(n)|}{|\mathbb{F}(m) \cup \mathbb{F}(n)|} \tag{4.7}$$

4.3.3 MCRM: Multi-dimensional Context-aware Recommendation Module

We now present the last module of *APIMatchmaker*, which performs multi-dimensional context-aware API recommendation after filtering our incompatible APIs.

4.3.3.1 Filtering out incompatible APIs

Recall that certain Android APIs are only available in a number of SDK versions. When developing apps with a dedicated Android SDK version, developers can only take advantage of the APIs available in that SDK. However, the resulting app is expected to be able to run different devices running different Android frameworks (i.e., SDK versions). This mismatch has led to the well-known fragmentation issue in the mobile community, for which Android apps may crash on certain devices while running smoothly on others. Typically, there are two types of compatibility issues introduced by the fast evolution of the Android framework.

- **Forward Compatibility Issue** implies that a given app developed targeting a given API level may not execute seamlessly on devices running Android with higher API levels.
- **Backward Compatibility Issue** implies that a given app developed with a given API level may not perform normally on devices running Android systems with lower API levels.

Because of the aforementioned compatibility issues, we have to take SDK versions into account when recommending possible APIs for implementing an active method. Fortunately, Android apps have been provided with a manifest file to configure the supported SDK versions.

- *minSdkVersion*, i.e., the minimum API Level at which the app is intended to run. This attribute is leveraged by Google Play to filter out devices with SDK platform versions lower than the value declared in *minSdkVersion*.

- *targetSdkVersion*, i.e., the API Level that the app is targeting. If this attribute is not set explicitly, the default value will be set to the value of *minSdkVersion*.

In this module, before running the multi-dimensional context-aware recommendation algorithm, we take additional efforts to extract the supported SDK versions from the apps in the training dataset and subsequently filter out such APIs that may cause compatibility issues (i.e., the incompatible APIs should not be included in the recommended API list). We achieve this function by automatically taking into account the lifecycle of APIs (i.e., when they are introduced and when they are excluded in the framework), which are extracted by checking the evolution of the Android frameworks, following the CiD approach introduced by Li et al. [91].

4.3.3.2 Multi-dimensional context-aware recommendation

Using our *APIMatchmaker*'s second module, the search space is reduced from all the methods of the training apps to a set of methods selected from a set of apps in the training set. In this last step, we introduce our multi-dimensional context-aware recommendation approach. This aims to recommend APIs for the method under editing in the active app project.

In particular, *APIMatchmaker* first determines the number of Android APIs (k) accessed by the selected apps (x) and then models them into a $(y + 1) * k$ matrix. As shown in Table 4.1, methods (selected ones $m_1 \rightarrow m_y$ plus the one under editing m_{edit}) are represented as rows, and APIs are represented as columns. For the elected y methods, each of their cells in the matrix will be set to either true (1) or false (0), representing whether the corresponding API has been accessed by the method or not. For example, cell (m_2, api_k) is

set to be 1, indicating that method m_2 has accessed api_k. For the method under editing (i.e., m_{edit} in the last row), each of its cells will be set to either true (1) or unknown (-1). The *true* cells indicate the set of APIs that have already been leveraged by the method under editing (i.e., m_{edit}). The *unknown* ones are the possible candidates that could be needed to complete m_{edit}. The goal of our *APIMatchmaker*'s last module is hence to predict which of the possible candidates could be used by the method under editing so as to recommend them to app developers to assist them in completing the method code.

Table 4.1: An example of the encoding matrix.

	api_1	api_2	api_3	api_4	api_5	...	api_k
m_1	1	1	0	1	0	1	1
m_2	1	1	0	0	0	1	1
m_3	1	1	0	0	1	1	0
...	1	1	1	0	0	0	1
m_y	1	1	1	1	1	0	1
m_{edit}	1	1	-1	-1	-1	-1	-1

Collaborative filtering has been often used for recommendation algorithms to implement recommendation systems [176]. A typical application of collaborative filtering is to recommend items that a user is most likely to purchase based on his/her past shopping records or information about other users with similar purchasing behaviors in an e-commerce system. Chen et al. [31] introduce the concept of context into the traditional collaborative filtering algorithm. The purpose of this is to recommend to the current user in the present context what other like-minded users do in a similar context.

Borrowing the idea of context-aware collaborative filtering for recommending items for users to purchase, in this work we leverage the same algorithm to recommend the candidate usage of Android APIs. In our work, the method plays the role of a **user**, the API plays the role of an **item** and the app project, to which the method belongs, plays the role of **context**. A **rating** (e.g., a numerical value) is further associated with a user and an item.

The prospective outcome of a collaborative filtering system is a set of predicted ratings (aka. recommendations) for a specific user and a subset of items [24]. The recommendation system considers the most similar users to the active user (aka. neighbours) to calculate new ratings. Additionally, based on the traditional collaborative filtering approach, we propose a new algorithm called multi-dimensional context-aware collaborative filtering. The idea behind this new algorithm is to integrate different types of similarity metrics to fulfill the API usage recommendation.

Let us define the project, i.e., context C as a tuple of z different types of similarity metrics, where $c_t(t \in 1...z)$ is a type of context.

$$C = (c_1, c_2, ..., c_z) \tag{4.8}$$

Except the APIs already included in the active method m_{edit}, for each api accessed by the x APKs, ***APIMatchmaker*** computes a score for each cell representing an api, i.e., cells set as -1 in the encoding matrix [3] shown in Table 4.1. The probability of recommending a given API api to m_{edit} can then be calculated via the following formula [31], where M is the set of the y most similar method signatures, sim_γ is defined by Formula 4.7, and $r_{\bar{m_{edit}}}$ and $r_{\bar{m}}$ are the mean ratings of m_{edit} and m, respectively.

$$p_{m_{edit},api,C_{m_{edit}}} = r_{\bar{m_{edit}}} + \frac{\sum_{m \in M}(R_{m,api,C_{m_{edit}}} - r_{\bar{m}}) \cdot sim_\gamma(m_{edit}, m)}{\sum_{m \in M} sim_\gamma(m_{edit}, m)} \tag{4.9}$$

In our implementation, $r_{\bar{m}}$ can be obtained by the encoding matrix in Table. 4.1, i.e., calculating the average rating of the cells in the row corresponding to m. For $r_{\bar{m_{edit}}}$, we set its value to 0.8 following the general practice of the state-of-the-art [119].

The weighted rating of each method $m \in M$, i.e., $R_{m,api,C_{m_{edit}}}$, with respect to API api and project context of the method under editing, can be calculated via the following

[3] The encoding matrices of different projects in the training set are essentially a simplified presentation of the context-aware 3-dimensional scoring matrix, as mentioned in FOCUS [119]. Interested readers are encouraged to read this paper for more details.

formula, where w_t is the weight we assign to each type of context and $\sum_{t=1}^{z} w_t$ should equal to one, and sim_t is defined by Formula 4.4 and Formula 4.5.

$$R_{m,api,C_{m_{edit}}} = \sum_{t=1}^{z} w_t \cdot r_{m,api,C_m} \cdot sim_t(C_{m_{edit}}, C_m) \tag{4.10}$$

We rely on the similarity of two contexts – two-dimensions: topic and code implementation – to recommend API usages. In this case, project context C will be (c_1, c_2), respectively representing the contexts of app topic and code implementation. Similarly, (w_1, w_2) will be respectively the weights of app topic and code implementation. These two weights can be configured by the users of our approach. By default, we set their values to be $(0.2, 0.8)$.

Output. The output of *APIMatchmaker* will be a list of Android APIs that are ranked based on the scores calculated via Formula 4.9. This list will be continuously updated to adapt to the change of the method under editing. Moreover, apart from the top-N APIs recommended based on the already written code in the active method, *APIMatchmaker* will also generate API usage samples that are extracted from the selected most similar apps, to help developers make use of the recommended APIs.

4.4 Experimental Setup

To evaluate the effectiveness of *APIMatchmaker*, we need to answer the following research questions:

- **RQ1:** How effective is *APIMatchmaker* in recommending accurate APIs for Android app developers to complete their development?
- **RQ2:** How does *APIMatchmaker* compare with other state-of-the-art tools?
- **RQ3:** To what extent do different parameters affect the performance of *APIMatchmaker*?

- **RQ4:** How effective is the multi-dimensional context-aware collaborative filtering approach applied by *APIMatchmaker*?

4.4.1 Dataset

To answer these research questions, we crawled a large set of real-world Android apps from Google Play, including their descriptions advertised on Google Play. Since it is not straightforward to crawl Android apps from Google Play, we resorted to collecting the latest Google Play apps from the well-known AndroZoo dataset, which has already been leveraged by various research projects [10]. At the moment, AndroZoo contains over 10 million Android apps crawled from various sources, including the official Google Play app store.

Since our collected dataset will be used as a training set for *APIMatchmaker* to learn API usages, it would be great if the collected apps have significant usage of different Android APIs. To this end, when crawling apps from AndroZoo, we checked API usage of each downloaded app and only retained the ones that have at least six (6) methods using at least fifteen (15) Android APIs.

Furthermore, when collecting apps from AndroZoo, we realize that not all the Google Play apps in AndroZoo are currently available on Google Play (e.g., those apps might be removed already), and not all the apps are described in English. As a result, we cannot obtain or parse those apps' descriptions, and thereby we have to ignore them when preparing our dataset. In addition, some apps, although different in terms of their hash digests (i.e., SHA256), are essentially the same app (i.e., share the same app package name but with different versions). Unfortunately, such apps share the same app description and thereby could introduce biases to our approach and experiments. To mitigate this, we only retain the latest app version in our dataset. Finally, because of some corner cases, some apps cannot be successfully parsed by our tool-chain to preprocess Android apps. Indeed, some

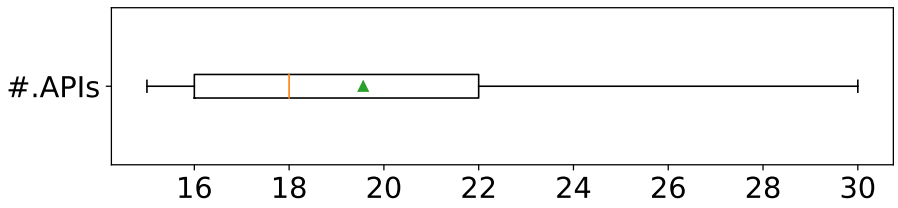


Figure 4.5: Distribution of the numbers of API in the methods selected from the 12,000 apps.

apps do not contain the AndroidManfest configuration file, so that their app package names cannot be extracted. In this work, we simply ignore those apps. Eventually, we stopped the collection at 12,000 apps from roughly 40,000 apps randomly selected from AndroZoo. Our final dataset is made up of 12,000 unique Android apps, along with their descriptions. Fig. 4.5 presents the basic statistics of the numbers of unique APIs called per method in the 12,000 apps. The median and average numbers are 18 and 19.8, respectively. Our final dataset is made up of 12,000 unique Android apps, along with their descriptions.

4.4.2 Experimental settings

Since it is hard to find active app projects under development, we use existing apps for simulation. Specifically, given an Android app with Δ methods declared, we propose to simulate its development at two different stages: **Stage 1:** App developers have completed the implementation of $\Delta/2 - 1$ methods and are now preparing to finish the $(\Delta/2)$-th method; **Stage 2:** App developers have completed the implementation of $\Delta - 1$ methods and are now preparing to finish the last method (Δ-th). In each stage, for the method under editing, we further take into account two scenarios: **Scenario 1:** The method has already accessed into one Android API; **Scenario 2:** The method has accessed into four Android APIs. Combining the stages with scenarios, we eventually set up four experimental settings to evaluate the performance of *APIMatchmaker* in recommending Android APIs for supporting the development of Android apps:

- **E1 (Stage 1, Scenario 1)**: The active app project has $\Delta/2 - 1$ method completed and the $\Delta/2$-th method under editing has already accessed into one Android API.
- **E2 (Stage 1, Scenario 2)**: The active app project has $\Delta/2 - 1$ method completed and the $\Delta/2$-th method under editing has already accessed into four Android APIs.
- **E3 (Stage 2, Scenario 1)**: The active app project has $\Delta - 1$ method completed and the Δ-th method under editing has already accessed into one Android API.
- **E4 (Stage 2, Scenario 2)**: The active app project has $\Delta - 1$ method completed and the Δ-th method under editing has already accessed into four Android APIs.

For each of these four settings, we use the standard procedure, i.e., 10-fold cross-validation, to evaluate the performance of our *APIMatchmaker*'s recommendations. To this end, we randomly divide our dataset (i.e., 12,000 apps) into ten sets (1,200 apps in each set). We then select nine sets to fulfill the training set and use the remaining set for testing. We repeat this process ten times to make sure that each of the ten sets has been regarded as a test set once. The average scores of the ten tests are then reported as the final performance of our approach. Since it is not easy to get the actual developing order of each Android project, the selection of the completed methods is random.

4.4.3 Evaluation Metrics

Given a method under editing in an active Android app project, the objective of our approach is to recommend a ranked list of API calls (e.g., N APIs) to help developers complete the implementation of the method. To help assess whether *APIMatchmaker* fulfills this objective, we leverage the following metrics to evaluate the effectiveness of our approach.

For each test sample, we only consider one active method. Given a set of projects P under testing, for the method under editing m in each project $p \in P$, *APIMatchmaker*

generates N recommended APIs, i.e., $R_N(p)$, to fulfill m.

Success rate: We consider that a recommendation is successful for project p as long as at least one out of the N APIs hit the Ground-Truth set $GT(p)$. The success rate for the $|P|$ projects can then be calculated via Formula 4.11, where $GT(p)$ stands for the set of APIs actually accessed by m in p, and $match_N(p)$ is defined as the intersection of the recommended N APIs and $GT(p)$, i.e., $match_N(p) = R_N(p) \cap GT(p)$.

$$success\ rate@N = \frac{count_{p \in P}(|match_N(p)| > 0)}{|P|} * 100\% \tag{4.11}$$

Precision and Recall: For each test sample, $precision@N$ is the ratio of the top N recommended APIs matching $GT(p)$, and $recall@N$ is the ratio of APIs belonging to $GT(p)$ falling in the top N recommendations.

$$Precision@N = \frac{|match_N(p)|}{N} * 100\% \tag{4.12}$$

$$Recall@N = \frac{|match_N(p)|}{|GT(p)|} * 100\% \tag{4.13}$$

4.5 Results

4.5.1 RQ1: Performance of *APIMatchmaker*

In this first research question, we investigated the performance and effectiveness of our approach *APIMatchmaker*. Following the settings described in Section 4.4.2, all the experiments are conducted based on the default parameters of *APIMatchmaker*, i.e., ten similar apps ($x = 10$), six similar methods ($y = 6$), 20% and 80% weights respectively for app topic and app code implementation($w_1 = 0.2, w_2 = 0.8$).

Table 4.2 summarizes our experimental results with respect to different situations when different numbers of APIs considered for evaluation. Specifically, we present our results in

five different situations: 1, 5, 10, 15, 20. For example, when Result@5 is concerned, we leverage the top-5 recommended APIs to calculate the performance (success rate, precision, and recall) of our approach. Generally speaking, the more APIs considered, the higher the success rate our approach can achieve. Indeed, if only one API is considered, our approach can already hit the correct API at over 50% of success rate. When increasing the number of recommended APIs to 20, the success rate can exceed 80%. In other words, by checking at most 20 APIs and for more than 80% of cases (i.e., methods under editing), app developers can successfully find the right APIs to complete the implementation of the method. This experimental result shows that our approach is effective in recommending APIs for supporting developers to implement Android apps.

Furthermore, when different scenarios – comparing E1 to E2, or E3 to E4 – are considered, *APIMatchmaker* will achieve different performance. As shown in Table 4.2, *APIMatchmaker* always achieves better performance in Scenario 2, compared with the results yielded in Scenario 1. This result is expected by us because Scenario 2 provides more known APIs than Scenario 1. Indeed, the more known APIs are provided, the more close neighbor methods can be located, and thereby the higher performance can be achieved.

Table 4.2: Success rate of 10-fold cross-validation on the 12,000 randomly selected apps for *APIMatchmaker*, the state-of-the-art tool FOCUS, and the baseline of our approach. Result@N indicates the number of recommended APIs considered for evaluation.

N	***APIMatchmaker***				***Baseline 1 - FOCUS***				***Baseline 2***			
	E1	**E2**	**E3**	**E4**	**E1**	**E2**	**E3**	**E4**	**E1**	**E2**	**E3**	**E4**
1	52.68%	60.34%	54.09%	61.1%	45.89%	51.92%	45.74%	54.01%	49.73%	49.71%	49.9%	49.1%
5	68.36%	77.33%	69.17%	78.08%	62.28%	71.14%	63.9%	72.43%	68.75%	66.99%	69.45%	66.58%
10	73.7%	82.8%	75.6%	83.04%	68.99%	77.41%	70.13%	78.45%	75.62%	73.27%	75.69%	73.48%
15	77.34%	88.03%	78.96%	88.36%	73.94%	83.13%	74.54%	84.31%	79.35%	78.4%	79.76%	79.39%
20	81.85%	91.61%	82.59%	91.64%	79.45%	87.43%	79.69%	89.14%	81.83%	81.84%	82.52%	83.1%

Even for the same scenario, when different stages are taken into account – Stage 1 to Stage 2 and comparing E1 to E3 and E2 to E4 – it is interesting to observe that the Stage 2 setting can always get a higher performance compared to that of the Stage 1 setting. This

suggests that the success rate increases when the number of completed methods increases. In other words, *APIMatchmaker* will be more useful for app projects that are already at a later stage.

We now go one step further to check the sensitivity of our approach to the size of the training dataset. Ideally, we would hypothesize that the larger the training dataset, the higher the performance our approach would achieve. Indeed, a larger set of training dataset could potentially allow *APIMatchmaker* to select x most similar apps that are even closer to the active project under development than selecting from a small dataset. To this end, apart from the 12,000 training apps considered in this work, we conduct two new experiments, each considering 3,000 and 6,000 apps as the training set, respectively, where these apps are randomly selected from the 12,000 datasets. Fig. 4.6 illustrates the experimental results. Clearly, in all the four experimental settings (cf. E1-E4), these results confirm our hypothesis that the performance of *APIMatchmaker* increases when more apps are considered for the training set.

Fig. 4.7 further compares the experimental results obtained by excluding the app topic context (i.e., only taking code implementation into consideration, i.e., $(w_1 = 0, w_2 = 1)$. The experimental results confirm our findings that the app topic context is helpful for finding the most similar apps. Interestingly, the benefit is even more significant when the size of the training set is small.

RQ1 Answer

With over 80% or even 90% (in scenario 2) success rate, *APIMatchmaker* achieves promising results in recommending API usages at Result@20, and yields acceptable results in an extreme case when only one recommended API is concerned. Furthermore, the more knowledge *APIMatchmaker* can learn from the active app project under development (in later stages), the higher performance *APIMatchmaker* can achieve in successfully recommending API usages.

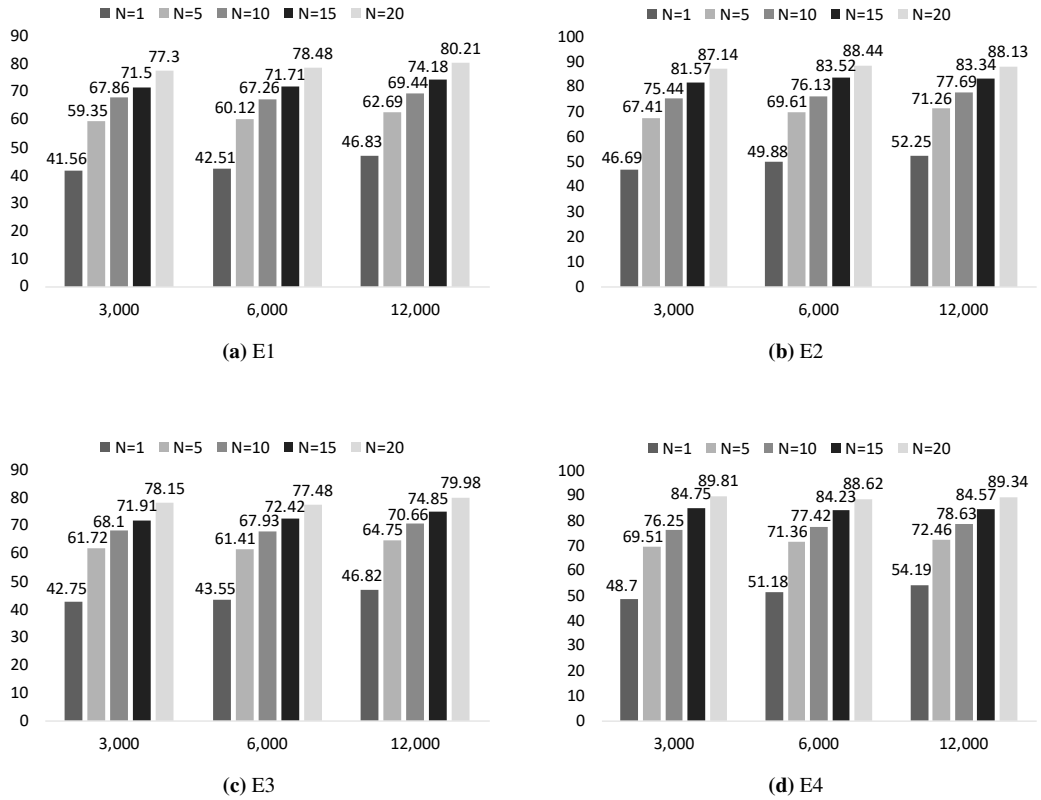


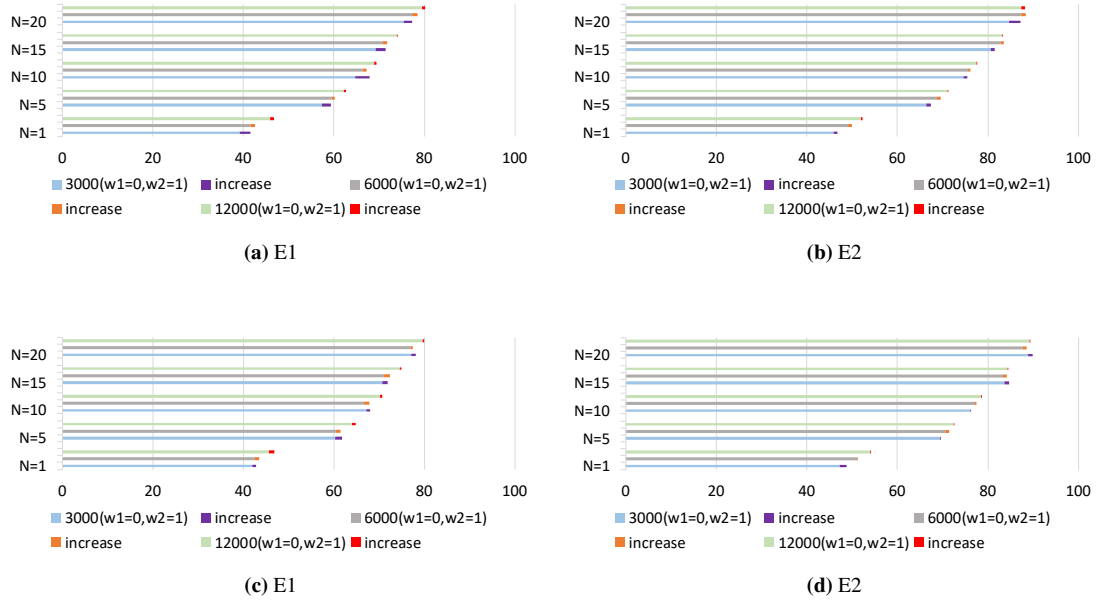
Figure 4.6: Experimental results observed by varying the size of the training dataset.

4.5.2 RQ2: Comparison with the State-of-the-art

We are interested in comparing the performance of our *APIMatchmaker* with that of other state-of-the-art approaches. Specifically, we compare our approach with the state-of-the-art FOCUS approach and a straightforward baseline approach.

***Baseline 1* - FOCUS:** FOCUS was initially designed to learn from open-source Java projects to recommend APIs for supporting the development. The authors have then extended their work to also support the analysis of closed-source Android apps [120]. In this work, we use their extended version to compare against our *APIMatchmaker* tool.

***Baseline 2*:** In order to further verify the effectiveness of *APIMatchmaker*, we construct a baseline approach based on probability statistics. The baseline approach works as



(a) E1 **(b)** E2

(c) E1 **(d)** E2

Figure 4.7: Success rate increases brought by the app topic context (i.e., the default setting, $w_1 = 0.2, w_2 = 0.8$) compared to the results only observed via code implementation context (i.e., $w_1 = 0, w_2 = 1$).

follows:

(1) Training Phase: For each app in the training set, we implement a static analyzer and leverage it to parse all the methods declared in the app. In each method, the static analyzer further extracts all its accessed Android APIs, along with their execution sequences. Then, based on the extracted information (API sequences), inspired by the idea of Function Call Graph raised by McMillan et al. [112], we construct a Weighted API Invocation Graph (WAIG). In WAIG, each Android API is recorded as a node, and API execution sequences are recorded as weighted edges. The weights of edges are simply set based on the number of times API execution sequences appear in the apps of the training set. For instance, suppose there is a method that has accessed two Android APIs (api_1 and api_2), and api_1 is called before api_2. In this example, the two APIs will be included as two nodes (will create new nodes if not already existed) in the WAIG. These two nodes will then be connected via a directed edge (i.e., $api_1 \rightarrow api_2$). If the directed edge already exists, there is no need to

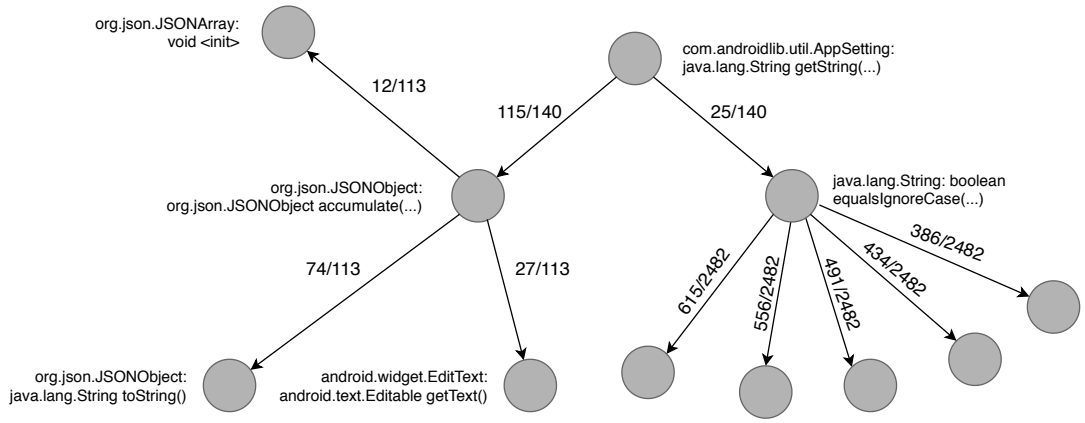


Figure 4.8: An example of the constructed weighted API dependency graph applied to the *Baseline 2* approach.

create the edge anymore but simply increase the weight by one to the existing edge. Fig. 4.8 portrays a sample WAIG (a sub-graph of the final WAIG built based on the training apps). In this case, "com.androidlib.util.AppSetting:java.lang.String getString()" is followed by "org.json.JSONObject: org.json.JSONObject accumulate()" or "java.lang.String: boolean equalsIgnoreCase()", but the former occurs more frequently, so the weight is higher than the latter.

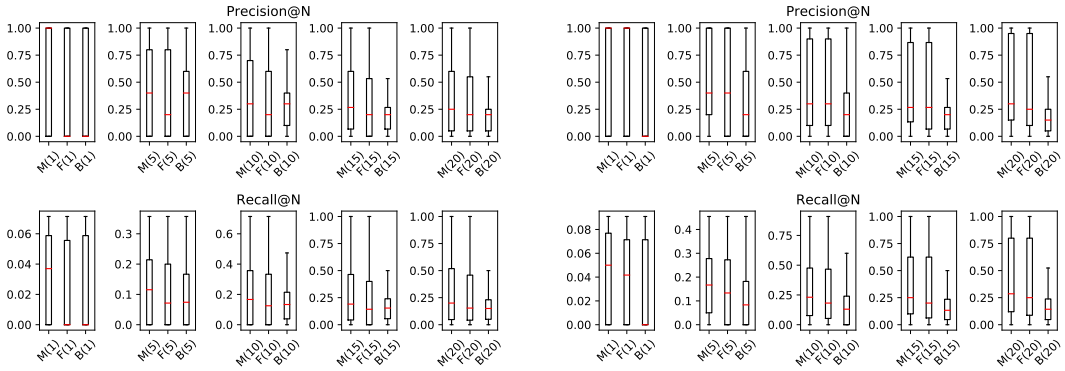
(2) Testing Phase: Given a method under editing and the APIs calls already written in it, our baseline approach will try to locate the same API execution sequences on the WAIG (built based on a set of training apps). If an exact match cannot be achieved, a similar execution sequence will be considered. Starting from the located sequence (e.g., the last node), the baseline approach will simply take its succeeded N-nodes (with edge's weights taken into account) as the list of APIs for the recommendation.

To enable a fair comparison, we use the same setting to evaluate the effectiveness of FOCUS and this baseline approach: 10-fold cross-validation on the randomly selected 12,000 APKs with the same training and test set applied in each fold. The experimental results

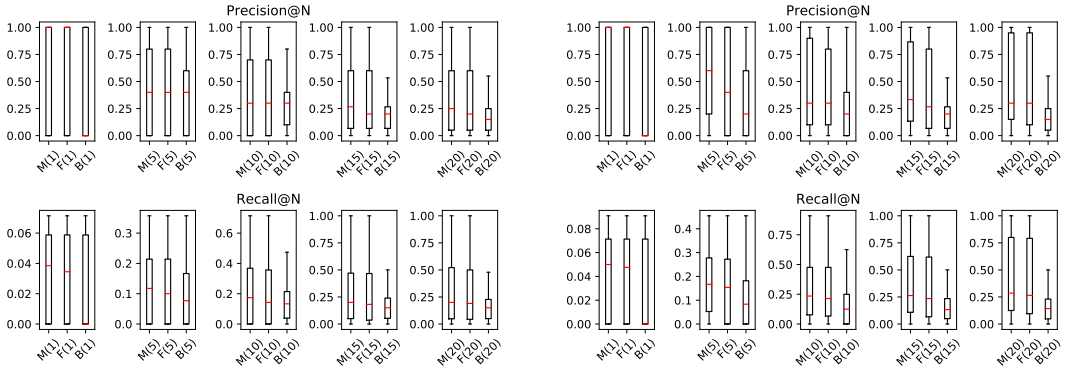
(i.e., success rate) of FOCUS and the baseline approach are shown in the last column of Table. 4.2. Generally, FOCUS and the baseline approach achieve more or less the same performance while *APIMatchmaker* outperforms both of them in all the experimental settings. Recall that FOCUS shares the same collaborative filtering algorithm with our approach, and its outputs, although lower, share the same pattern as well, i.e., the results of E2/E4 are much higher than that of E1/E3. This pattern, however, does not appear in the simple baseline approach, for which the experimental results are more or less the same across all the four settings. If only comparing the results of FOCUS and the baseline approach, FOCUS yields even lower performance when E1/E3 settings (with only one API written) are considered. This evidence further confirms our previous finding that the more knowledge our approach (or similar approach) can learn from the method being editing, the higher performance it can achieve. Moreover, the simple baseline approach could also be a suitable supplement for the complex learning-based approaches, especially when there is no sufficient pre-knowledge to supervise the learning.

Fig. 4.9 shows the distribution of precision and recall scores of *APIMatchmaker*, FOCUS, and our baseline approach, in the pre-defined four experimental settings (i.e., E1, E2, E3, and E4), respectively. In all experimental settings, both in terms of precision and recall, *APIMatchmaker* outperforms the two baselines. Mann-Whitney-Wilcoxon (MWW) tests additionally confirm that the differences between *APIMatchmaker* and the two counterparts are statistically significant. As highlighted in Table. 4.3, which summarizes the $p-values$ of MWW tests conducted between *APIMatchmaker* and *FOCUS*, as well as between *APIMatchmaker* and the baseline approach, the $p-values$ are always smaller than *0.005*[4]. The only exceptions are the comparison results in E2, E3 and E4 between *APIMatchmaker* and *FOCUS* when Top-20 APIs are considered.

[4]Given a significance level $\alpha = 0.005$, if $p - value < \alpha$, there is five chance in a thousand that the difference between the two datasets is due to a coincidence.



(a) E1 **(b)** E2



(c) E3 **(d)** E4

Figure 4.9: Distribution of precision and recall for *APIMatchmaker*, *FOCUS* (*i.e., Baseline 1*) and the *Baseline 2* of our approach.

RQ2 Answer

Under the same experimental settings, *APIMatchmaker* outperforms both *FOCUS* (i.e., *Baseline 1*) and *Baseline 2* in achieving significantly higher success rate, precision, and recall for recommending APIs for supporting the development of Android apps.

4.5.3 RQ3: Impact of parameter tuning on *APIMatchmaker*

We now explore the impact of altering the parameter values related to the number of similar apps and methods on the performance of *APIMatchmaker*. To this end, we designed

Table 4.3: The $p - values$ of Mann-Whitney-Wilcoxon Tests on the comparison results of between *API-Matchmaker* and Baseline 1 - *FOCUS*, as well as between *APIMatchmaker* and Baseline 2.

	E1				**E2**			
	FOCUS		Baseline		*FOCUS*		Baseline	
N	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	5.14E-18	6.73E-18	4.68E-4	1.93E-3	5.17E-22	1.26E-22	6.08E-44	1.09E-40
5	1.04E-23	1.16E-24	6.64E-60	6.29E-52	8.28E-17	1.65E-18	3.48E-255	2.19E-229
10	1.04E-20	1.05E-21	4.41E-73	2.84E-72	1.09E-13	1.18E-14	2.23E-308	2.23E-308
15	8.37E-15	1.86E-15	6.41E-85	1.39E-83	3.65E-7	1.71E-7	2.24E-230	6.79E-234
20	3.87E-6	2.46E-6	6.29E-51	5.16E-52	1.99E-2	1.84E-2	1.68E-111	4.11E-113

	E3				**E4**			
	FOCUS		Baseline		*FOCUS*		Baseline	
N	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
1	1.04E-6	4.44E-7	6.33E-8	5.17E-7	1.96E-3	5.01E-3	3.85E-51	2.62E-45
5	9.97E-8	5.3E-8	5.34E-65	1.45E-55	3.88E-6	2.39E-6	4.41E-290	1.19E-257
10	4.24E-7	3.64E-7	1.12E-92	1.3E-90	1.42E-5	1.51E-5	2.23E-308	2.23E-308
15	6.19E-6	5.96E-6	1.47E-113	1.98E-112	7.34E-4	7.2E-4	9.81E-263	1.53E-265
20	8.04E-3	8.31E-3	1.04E-60	1.577E-61	5.02E-2	5.13E-2	9.25E-120	1.07E-121

multiple sets of experiments (considering different numbers of most similar projects and methods) to fulfill this objective. Specifically, we set $x = 5, 10, 20$ and $y = 3, 6, 12$. In total, we conduct nine (i.e., $3*3$) groups of experiments. The other parameters (i.e., context weights) are kept to the default value pre-configured in *APIMatchmaker*.

Table 4.4 presents the experimental results for the pre-defined four settings, respectively. Similar to our previous findings, no matter which group of experiments is concerned, when increasing the number of APIs (i.e., N) to be considered, the performance increases continuously. Furthermore, when looking at the increases in the number of most similar apps (with the same number of most similar methods), or the number of most similar methods (with the same number of most similar apps), in most of the cases, the success rate of *APIMatchmaker* also increases constantly. Indeed, with the largest number of most similar apps and methods ($x = 20$, $y = 12$), *APIMatchmaker* achieves the best performance in all the four settings, giving over 83% of success rate when only one API is known and at least 92% of success rate when four APIs are known.

We further explore the impact of increasing the number of most similar apps and meth-

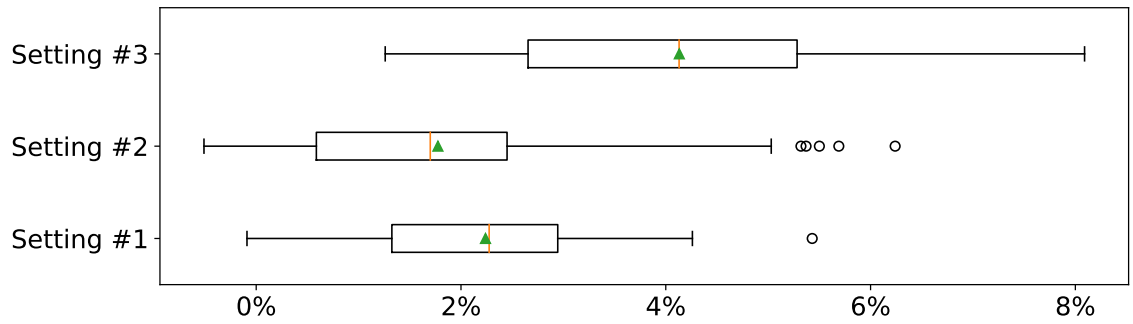


Figure 4.10: Distribution of the performance increases by varying the size of most similar apps and methods (i.e., parameters x or y).

ods. Specifically, given a group of parameters (x, y), we would like to check which of the following two settings (i.e., (2x, y) and (x, 2y)) contributes more to the increase of the performance of *APIMatchmaker*, and the results of (2x, 2y) setting are also analysed for reference. Fig. 4.10 illustrates the comparison of the improvements brought by the previous three settings: Setting #1: Double the number of most similar apps (2x, y), Setting #2: Double the number of most similar methods (x, 2y). Setting #3: Double the number of most similar apps and methods (2x, 2y). The improvements will be the difference between the new results and the original ones, i.e., (2x, y) - (x, y), (x, 2y) - (x, y) and (2x, 2y) - (x, y) respectively. Clearly, the improvements brought by increasing parameter x are more significant than that of increasing parameter y, and the performance of *APIMatchmaker* will be more significantly improved, if both x and y are increased, as indicated by the results of Setting #3. This significance has further been backed up by the MWW testing result.

RQ3 Answer

Generally, the more number of most similar apps, or the more number of most similar methods considered, the higher performance *APIMatchmaker* will yield. Nevertheless, the former case (i.e., increasing the number of most similar apps) contributes more to the performance of *APIMatchmaker* than the latter case (i.e., increasing the number of most similar methods).

Table 4.4: Success rate of 10-fold cross-validation on the 12,000 randomly selected apps by varying the size of most similar apps and methods (i.e., parameters x and y).

					E1				
N	**(5, 3)**	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**
1	46.83%	47.68%	50.26%	49.01%	52.68%	54.83%	51.31%	55.19%	59.1%
5	62.69%	65.2%	68.03%	64.49%	68.36%	70.61%	65.02%	69.27%	72.52%
10	69.44%	71.3%	74.24%	70.38%	73.7%	76.74%	70.5%	74.34%	77.63%
15	74.18%	74.92%	77.73%	74.38%	77.34%	79.92%	74.53%	77.59%	80.51%
20	80.21%	80.31%	81.56%	80.76%	81.85%	83.41%	80.74%	81.48%	83.96%
					E2				
	(5, 3)	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**
1	52.25%	55.02%	55.98%	57.75%	60.34%	61.67%	59.46%	62.27%	64.93%
5	71.26%	74.12%	75.43%	73.78%	77.33%	78.33%	74.11%	77%	79.29%
10	77.69%	80.18%	81.45%	79.49%	82.8%	83.71%	80.16%	82.91%	85.15%
15	83.34%	85.34%	86.53%	84.91%	88.03%	88.26%	85.27%	87.83%	89.61%
20	88.13%	89.4%	90.17%	88.44%	91.61%	91.52%	88.92%	91.71%	92.87%
					E3				
N	**(5, 3)**	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**
1	46.82%	49.39%	50.94%	50.29%	54.09%	55.97%	52.01%	55.57%	60.1%
5	64.75%	66.44%	69.48%	65.24%	69.17%	71.68%	65.99%	69.87%	73.89%
10	70.66%	72.62%	74.9%	71.34%	75.6%	77.03%	71.67%	75.45%	78.83%
15	74.85%	76.48%	78.53%	75.3%	78.96%	80.29%	75.62%	78.45%	81.48%
20	79.98%	81.23%	82.54%	80.29%	82.59%	83.9%	80.52%	82.12%	84.48%
					E4				
	(5, 3)	**(10, 3)**	**(20, 3)**	**(5, 6)**	**(10, 6)**	**(20, 6)**	**(5, 12)**	**(10, 12)**	**(20, 12)**
1	54.19%	55.73%	56.34%	58%	61.1%	62.58%	60.67%	63.54%	65.78%
5	72.46%	74.84%	76.17%	74.19%	78.08%	78.99%	75.88%	78.38%	80.6%
10	78.63%	80.76%	82%	80.36%	83.04%	84.41%	81.65%	83.92%	86.17%
15	84.57%	86.26%	87.19%	85.67%	88.36%	89.18%	87.47%	89.1%	90.74%
20	89.34%	89.66%	90.65%	89.66%	91.64%	92.26%	90.98%	91.97%	93.25%

4.5.4 RQ4: Effectiveness of the multi-dimensional context-aware collaborative filtering approach

Recall that we provide four parameters (x apps, y methods, (w_1, w_2) context weights) for users to customize the performance of *APIMatchmaker*. The previous section explores the sensitivity of *APIMatchmaker* on the number of similar apps and methods (i.e., x and y) to the final performance of recommending API usages to Android developers. Here we explore the impact of the other two parameters (i.e., w_1, w_2) on the recommendation performance of *APIMatchmaker*. Specifically, we are interested in evaluating the effectiveness of the multi-dimensional context-aware collaborative filtering algorithm applied

Table 4.5: Success rate of 10-fold cross-validation on the 12,000 randomly selected apps by varying the weights of the two contexts (i.e., app topic w_1 and code implementation w_2).

					E1					
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	45.89%	46.01%	**46.83%**	46.45%	46.14%	45.74%	44.92%	45.01%	45.65%	44.13%	39.64%
5	62.28%	62.3%	62.69%	62.29%	**63.3%**	63.2%	63.22%	63.12%	62.77%	61.74 %	54.97%
10	68.99%	69.12%	69.44%	**70.35%**	70.21%	70.11%	69.48%	68.74%	68.91%	66.35%	61.72%
15	73.94%	74.02%	**74.18%**	74.16%	74.15%	73.9%	73.73%	74.2%	74.24%	73.4%	67.06%
20	79.45%	79.68%	**80.21%**	80.1%	80%	79.75%	78.34%	78.99%	79.79%	77.35%	72.76%
					E2					
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	51.92%	52.12%	52.25%	52.21%	52.1%	52.46%	**52.52%**	52.15%	52%	51.75%	45.71%
5	71.14%	71.2%	71.26%	71.65%	**71.69%**	71.45%	71.17%	71.03%	70.64%	68.16%	62.33%
10	77.41%	77.51%	**77.69%**	77.49%	77.45%	77.4%	77.5%	77.33%	77.43%	69.76%	69.08%
15	83.13%	83.25%	83.34%	83.3%	83.28%	83.3%	83.36%	83.44%	**83.48%**	82.57%	75.17%
20	87.43%	87.74%	88.13%	**88.14%**	87.95%	87.63%	87%	87.56%	87.48%	86.1%	81.17%
					E3					
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	45.74%	45.89%	46.82%	46.73%	**46.89%**	46.73%	46.54%	46.44%	46.72%	45.16%	39.88%
5	63.9%	64.2%	**64.75%**	64.16%	63.98%	63.47%	62.79%	62.84%	63.65%	62.58%	56.98%
10	70.13%	70.25%	**70.66%**	70.33%	70.12%	70.1%	69.21%	70.12%	70.25%	68.16%	64.09%
15	74.54%	74.68%	74.85%	74.77%	**74.93%**	74.14%	73.58%	74.69%	74.77%	73.26%	68.33%
20	79.69%	78.5%	**79.98%**	79.43%	79.91%	79.21%	78.4%	78.98%	79.24%	77.37%	74.5%
					E4					
N	**(0,1)**	**(0.1,0.9)**	**(0.2,0.8)**	**(0.3,0.7)**	**(0.4,0.6)**	**(0.5,0.5)**	**(0.6,0.4)**	**(0.7,0.3)**	**(0.8,0.2)**	**(0.9,0.1)**	**(1,0)**
1	54.01%	54.12%	54.19%	**54.2%**	53.22%	53.87%	54.07%	53.89%	54.04%	53.74%	46.79%
5	72.43%	72.44%	**72.46%**	72.33%	71.95%	72.15%	72.02%	72.1%	72.18%	70.35%	63.29%
10	78.45%	78.55%	**78.63%**	78.41%	78.21%	78.37%	78.57%	78.46%	78.4%	77.25%	70.35%
15	84.31%	84.44%	**84.57%**	84.41%	84.21%	84.46%	84.52%	84.1%	84.02%	83.2%	77.19%
20	89.14%	89.2%	89.34%	**89.4%**	88.24%	89.17%	88.76%	88.21%	87.91%	86.53%	83.19%

by *APIMatchmaker*. Specifically, since we only take two-dimensional data (app code and app topic) into consideration, we evaluate the sensitivity of the weights we have assigned to these two dimensions. By default, the weights for app topic and code (w_1, w_2) are set to be (0.2, 0.8). In this work, we compare this default setting with another ten settings formed by altering the weights, i.e., (0, 1), (0.1, 0.9), (0.3, 0.7), (0.4, 0.6), (0.5, 0.5), (0.6, 0.4), (0.7, 0.3) (0.8, 0.2), (0.9, 0.1) and (1, 0). Weights (0, 1) and (1, 0) respectively stand for the cases where only app code or app topic is considered for locating similar methods in the training set. All the other parameters of *APIMatchmaker* are kept the same to ensure a fair comparison.

Table. 4.5 summarizes the experimental results. As shown in the seventh and thirteenth

columns, *APIMatchmaker* yields significantly worse results when only app topic is considered, showing that the app code is a very important dimension for locating similar apps in learning and recommending API usage patterns. Nevertheless, by considering only the app code, *APIMatchmaker* does not achieve the best performance either. Indeed, in almost all the cases, the combination of app topic and code (e.g., (0.2, 0.8), (0.3, 0.7), (0.4, 0.6)) achieves better performance than that of app code alone (0, 1) or app topic alone (1, 0). This empirical evidence shows that our multi-dimensional context-aware collaborative filtering algorithm is indeed effective and useful for learning API usages from large-scale Android apps.

RQ4 Answer

Balancing the weights of two types of context can indeed achieve better performance than considering one type along. This empirical evidence demonstrates that our multi-dimensional context-aware approach is effective and useful in finding the closest apps for learning API usages and subsequently allowing our approach to recommending more suitable APIs for methods under editing.

4.5.5 Artifact and Data Availability

To foster open science, we have made available online our full implementation and the relevant datasets on the famous open-access repository Zenodo [11], aiming to help readers reproduce our experimental results. We further put the full implementation as an open-source project on Github (i.e., the subsequent updates will hence be also recorded and opened) via the following link.

`https://github.com/SMAT-Lab/APIMatchmaker`

4.5.6 Threats to Validity

The validity of our study may have been impacted by several threats below we highlight the key ones.

Threats to External Validity. One of the main threats to external validity lies in the random selection of our training dataset, which may not be generic and representative of all the apps available in the ecosystem. We have attempted to mitigate this threat by selecting more than 10,000 real-world apps that have been all released over the official Google Play store. Additionally, we also evaluate the effectiveness of our approach by varying the size of the training dataset. The fact that there is no significant difference observed by doing that shows that this threat to the external validity of our study could be small. Furthermore, at the moment, we do not take into account app obfuscation in this work, despite it has likely been applied to some of the apps in our training dataset. Nonetheless, since we are mainly interested in learning API usages, which would not be impacted by simple obfuscation strategies such as method renaming, our approach should remain valid and effective. Indeed, as revealed by Dong et al. [40], the majority of Android apps are only obfuscated via simple strategies. Advanced strategies such as altering the code implementation are rarely adopted by real-world Android apps.

We leverage the app description on Google Play to represent the app topic. We consider these descriptions are reliable because they are publicly advertised by the app owners. However, the app descriptions may not be well-written or representative of the actual app topic. Ideally, it would be great if automated approaches could be introduced to validate the app descriptions before applying to our approach. This is nonetheless out of the scope of this work. We take it as our future work.

The implementation of *APIMatchmaker* relies on two static analysis frameworks, Soot [76] and JavaParser[5], for which their reliability issues could propagate to *APIMatchmaker* so as to threaten the validity of our approach. Nevertheless, both Soot and JavaParser are well-known frameworks that have already been shown practical and useful by many of our fellow researchers [15, 83, 36]. We hence believe the threat brought by these two tools

[5]https://javaparser.org

should not be significant.

So far, our *APIMatchmaker* only supports API recommendation for such app projects that are developed in Java, although there is nowadays a significant portion of Android apps developed using Kotlin [106], a new programming language introduced by Google to develop Android apps. Nonetheless, since Kotlin is designed to interoperate fully with Java, its syntax is quite similar to that of Java. Hence, it should be relatively straightforward to extend our approach to also support Kotlin-based app projects. We take this as our future work.

Threats to Internal Validity. The main threat is that we employ simulated experimental scenarios for evaluation rather than user studies. Drawing on the related work [119], we mitigate the threat by forming large-scale datasets with apps randomly collected from AndroZoo [10] and adopting 10-fold cross-validation to reduce the impact of contingency. We also attempt to simulate different development scenarios (at an earlier or later stage of the development) and examine the recommended results of *APIMatchmaker* with the ground-truth composed of the real APIs invoked by the method under editing. Nevertheless, we believe user studies are also necessary towards understanding if our approach has fulfilled the actual demand of Android app developers.

Since it is hard to guess and thereby simulate the sequence of methods developed by app developers, in our work, we resort to random selections to prepare the completed methods to fulfill our experiments. However, in the actual situation, the part that the developer has completed is usually related to the method under editing. As a result, the experimental results presented in this work may not reflect the actual capability of *APIMatchmaker* in practice. In order to explore the influence of the order of the methods belonging to the completed part of the test method on the experimental results, we conduct supplementary experiments by comparing the methods selected via a random order with methods selected based on their lexicographic order on the 12,000 apps. The final results suggest that the

impact seems to be marginal. Indeed, the $p - values$ of MWW tests are always larger than 0.05, indicating that there is no statistically significant difference between the aforementioned two experiments.

4.6 Discussion

To the best of our knowledge, *APIMatchmaker* is the first work that combines both code implementation and app topic text to learn and recommend API usage. Unlike most state-of-the-art works, which mainly focus on recommending Android APIs based on known *Java objects*, in *APIMatchmaker* we aim to give suggestions without knowing such *objects* information. Below we discuss the performance sensitivities of our approach with respect to different experimental settings.

Impact of the Training Dataset. As experimentally demonstrated in Section 5.4.1, when varying the training dataset size (e.g., in addition to the data set of 12,000 apps, we conducted two sets of comparative experiments on 6,000 and 3,000 datasets, respectively), the performance of *APIMatchmaker* also varies. Generally speaking, the larger the training dataset, the higher performance *APIMatchmaker* could achieve, as a larger training dataset could potentially allow *APIMatchmaker* to select most similar apps that are even closer to the active project under development than selecting from a small dataset. As also confirmed via experiments, the app descriptions (or topics) are also helpful for identifying the most similar apps. Nevertheless, in this work, we have not evaluated the impact of app qualities (neither the apps' code implementation nor their descriptions) selected for training the model. This, however, is outside of the scope of this work. We hence consider it as our future work.

Impact of the number of known accessed APIs in the active method under editing. Recall that in all the experiments conducted in answering the proposed research questions, we directly follow the settings of FOCUS and consider that the method under editing has al-

ready accessed four Android APIs, as described in Scenario 2 of E2 and E4 in Section 4.4.2. We now explore the influence of the numbers of APIs that have already been accessed by the method under editing on the recommendation performance of *APIMatchmaker*. Ideally, when the number of known accessed APIs increases (as the developer continuously edits the active method), the recommended results should be improved as well, as more information becomes available for training. Towards confirming this hypothesize, we conduct another round of experiments by replicating the E2 and E4 experiments with the number of known APIs changed, e.g., from four to half of the APIs leveraged by the active method. Our experimental results indeed confirm the previous hypothesis, i.e., the precision and recall are improved when half of leveraged APIs (rather than four) are considered. This experimental evidence further demonstrates the usability of our recommendation approach.

On the Importance of taking Android SDK versions into account. We would like to emphasize that the SDK version check of the recommended APIs is critically important since the evolution of the Android framework could cause compatibility issues. For example, API <*SentenceSuggestionsInfo: int getLengthAt(int)*> is added in API level 16 while the min sdk version of app *com.college.theking.christ* is 15. When the above API is recommended for the implementation of the app, compatibility issues will occur if the SDK version supported by the API is not checked and known to the developer. In other words, if the developer does not protect the API with condition checks, when the app runs on an Android device with the SDK version 15, it will crash. In order to provide developers with more help, if such an API appears in our recommended list, *APIMatchmaker* will remove it and inform the developer of the reason for the removal, which is to separately remark it according to the incompatibility compared to the SDK versions of the app currently being developed. As for future work, instead of excluding incompatible APIs, we would like to enhance our recommendation approach to provide better options to resolve this problem, including alternative ways to bypass the compatibility issues (e.g., by recommending its

possible replacements).

Learning API usages from more fine-grained similar artifacts. In this work, to fulfill our multi-dimensional context-aware approach for recommending API usages, we have leveraged app descriptions to locate similar apps, aiming at learning API usages from apps that share similar functions with the one under development. Our experimental results have demonstrated that this information is indeed useful for improving the recommendation results. However, similarity analysis at the app level might be too coarse. More fine-grained similarity analysis could further improve the recommendation results. For example, one could leverage the methods' comments to compute method-level similarities when selecting relevant implementations for learning API usages. The methods' comments, however, are only available in source code projects, which are not the focus of this work. We hence regard it as our future work and encourage our fellow researchers to collaboratively explore this research direction.

4.7 Related Work

We summarize key related work from three aspects, i.e., recommendation in Android and in the field of software engineering, and collaborative filtering applied in software engineering.

4.7.1 Recommendation in Android Development

Since the development process of mobile apps relies heavily on API frameworks and libraries, in the Android community, researchers have devoted much effort to supporting Android API recommendation to better support mobile app development.

Some works try to give appropriate recommendation suggestions on third-party libraries [109, 97, 58], others knuckle down to the code level, that is, giving real-time suggestions during app development. Hence, many works have been done to extract pa-

rameters as recommendations in similar programming scenarios [135, 86]. Except for recommending third-party libraries or parameter values for APIs, there are a large number of researchers focusing on recommending Android APIs and their usage patterns. According to Wu et al. [168]'s defined categories, we introduce state-of-the-art research works related to API recommendations. Most integrated development environments (IDEs) haven been widely equipped with fundamental code completion features, which have been shown effective and useful by developers. Through the basic recommendation features of IDEs, developers can promptly complete an API by typing 'dot' subsequent an object instance to obtain a recommendation list generated based on the static information of the Android app under development.

At present, most of the advanced API recommendation mechanisms predict API usage patterns that are generated together with a given object instance. For example, Nguyen et al. [121] propose a sequence-based tool named DroidAssist to perform code completion for method calls based on Hidden Markov model of API usages (HAPI). They subsequently provide another approach, the key component of which is also HAPI, to train a statistical model of API usage from Dalvik Virtual Machine (DVM) bytecodes [122]. The objective of these approaches is to recommend the next method call as well as a more suitable method sequence. Users are required to provide the object instances being edited as query, which is not the same as the usage scenario of our approach. In our circumstance, even if the programmer does not enter an instance of the object, *APIMatchmaker* can also give suitable recommendations for the method currently under development.

Clustering techniques have also been employed to object usage-based Android API recommendation. Niu et al. [124] mine API usage patterns by clustering the data based on the associations of object usages, i.e., API sequences on a given class, while building the "gold set" manually based on human programming knowledge is time-consuming and a potential threat to the construct validity in terms of both establishment and evaluation.

In contrast, *APIMatchmaker* utilizes the context information to extract API usage patterns from similar apps, i.e., the construction is completed without human intervention. Yuan et al. [178] initially focus on the need to recommend event callbacks in the environment of Android application development and introduce an approach to recommend both functional APIs and the event callbacks that need to be overridden. Later on, the authors extend their work by establishing Android-specific API description databases designating the associations among diverse functionalities and APIs [179]. Unlike their work, which is based on structured and fixed databases, *APIMatchmaker* learns from a training set that can be easily adjusted to fulfill different requirements.

Code search, as another research topic in software engineering, has also been leveraged as a means to recommend API usages for Android developers. Indeed, with the objective of code search, Gu et al. [54] introduce an approach to generate API usage sequences based on a natural language query through a deep learning-based approach. Similarly, Jiang et al. [69] generate recommended code snippets based on multi-aspect features, including text, topic, and the number of lines, etc. Different from the aforementioned code search methods, Nie et al. [123] propose an approach leveraging knowledge learned from Stack Overflow to grow the performance of code search algorithms. All the above three works are mainly based on querying natural languages to perform relevant-API recommendations, which are naturally different approaches compared to ours. Nevertheless, we believe those approaches, together with ours, could supplement each other and hence be combined to better serve app developers.

The work most close to ours is the one proposed by Nguyen et al. [119], who introduce a context-aware collaborative filtering based algorithm to recommend Java method invocations utilizing Rascal M^3 model [6]. Later, the authors extend their work by leveraging the algorithm to the Android platform [120]. In our work, we extend the context to

[6]https://www.eclipse.org/jdt/core/

multi-dimensions to make full use of other features besides code implementation.

4.7.2 Recommendation in software engineering

Researchers have consecrated many efforts in offering fundamental recommendation features as a primary recommendation for modern IDEs [126, 146, 127]. In recent years, concerning further improve the efficiency of developers, advanced research works based on recommendations are emerging [32, 154, 181, 18, 46, 118, 140, 116]. As another example, Gu et al. [53] present a graph kernel-based approach to the selection of API usage examples by representing source code as object usage graphs.

Mcmillan et al. [112, 113] utilize graph-based matching to retrieve and visualize associated functions and their usages. Chan et al. [30] propose an optimized algorithm to search in an API graph with the given text query phrases. Thung et al. [155] take as input a textual description of a feature request and recommend API methods based on the similarity between textual API descriptions. Rahman et al. [134] exploit the keyword-API association identified by crowd-sourced knowledge of StackOverflow to enrich the translation between natural language query and code search. Raghothaman et al. [133] propose a work to suggest code snippets given API-related natural language queries by learning structured API call sequences from open-source code repositories. Huang et al. [64] propose BIKER, which leverages both Stack Overflow posts and API documentation to extract candidate APIs for ranking with a programming task described in natural language. The above works frequently mention the utilization of text as query phrases. However, in our work, the topic text of apps is uniquely used as a type of context together with code implementation, which is significantly different from the traditional modes of code search with text queries.

4.7.3 Collaborative filtering applied in software engineering

Collaborative filtering techniques are widely utilized in software engineering in general to implement recommendation systems. To assist developers in taking advantage of available

third-party libraries, Thung et al. [153] combine association rule mining and user-based collaborative filtering and propose a technique to recommend likely relevant libraries according to the libraries an application currently uses. Furthermore, Yu et al. [176] introduce an approach that combines collaborative filtering and Latent Dirichlet Allocation (LDA) to provide suggestions about third-party libraries for mobile apps. Moreover, He et al. [58] design a novel approach leveraging Matrix Factorization (a classic collaborative filtering based prediction approach) for predicting useful third-party libraries. Similarly, Xia et al. [171] employ an approach that combines a Matrix Factorization based latent factor model with a neighborhood-based method to capture implicit relations for improving the code reviewer recommendation, which has been acknowledged to be of great importance for software quality assurance. That is because due to the complexity of expertise and availability of candidate reviewers, it can be quite challenging to find appropriate reviewers.

Collaborative filtering techniques have also been introduced to recommend sampling methods to improve the performance of software defect prediction. For example, Sun et al. [149] present a collaborative filtering based sampling method recommendation algorithm to automatically suggest appropriate sampling methods for newly identified defect data.

4.8 Conclusion

In this paper, we have proposed a novel multi-dimensional context-aware collaborative filtering-based approach, *APIMatchmaker*, for recommending API usages to Android app developers. *APIMatchmaker* initially leverages both pure code implementations and app topics to identify high-level features and subsequently locate similar projects as well as methods that are closest to the active project's method under editing. Then, it utilizes the encoding matrix and rating algorithm to obtain the output of recommended APIs and extracts the code snippets (as API usage samples) from the original APKs files. Our experimental results on large-scale datasets demonstrate that *APIMatchmaker* is effective in

learning and recommending API usages for Android developers and is able to outperform both state-of-the-art and our baseline approaches.

As key areas for future work we plan to integrate our approach into Android Studio (the default IDE recommended for app developers to implement Android apps) as a plugin and aim at continuously recommending API usages during the whole development phase of given Android apps. The recommended list of API usages as well as their sample code, will be continuously updated based on the code written by app developers. In addition to recommending API usages for Android projects developed in Java, we also plan to support API recommendation for Kotlin-based Android apps.

Chapter 5

Towards Automatically Repairing Compatibility Issues in Published Android Apps

Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards automatically repairing compatibility issues in published Android apps. In Proceedings of the 44th International Conference on Software Engineering (ICSE '22). Association for Computing Machinery, New York, NY, USA, 2142–2153. https://doi.org/10.1145/3510003.3510128

5.1 Introduction

The heavy fragmentation problem of Android – many different Android versions (official and customized) running on different devices released by hundreds of manufacturers – has caused severe compatibility issues for the Android ecosystem. Android phone users often find that certain apps cannot be installed on their devices or can be installed but will crash later on if a specific function is reached, leading to poor user experiences. Actually, as revealed by Byron Muhlberg, there are over a billion Android devices no longer supported by Google. Hence, the users of those devices will likely encounter compatibility issues, especially when they want to leverage the latest Android apps, leading to serious problems in the mobile ecosystem.

To address this problem, approaches have explored various ways to automatically detect compatibility issues in Android apps [73, 90, 137, 60, 22, 188, 173, 144, 44, 56, 55, 77].

For example, Wei et al. [164] empirically looked into a set of fragmentation-induced compatibility issues (including those introduced by third-party manufacturers) in open-source Android apps. They further proposed a tool called FicFinder to detect the previously characterized compatibility issues automatically. Li et al. [91] proposed a generic approach called CiD, which detects API-related compatibility issues based on Android API lifecycle knowledge mined from the official's historical evolution Android framework.

Unfortunately the majority of these works only focus on detecting some compatibility issues, leaving many identified issues still unfixed in real-world Android apps. Updating incompatible APIs is a time-consuming endeavor and app developers are also known to be reluctant to repair their apps for fixing issues yielded by static analyzers [48]. App developers have to learn the usages of new APIs in order to replace the incompatible ones while maintaining backward compatibility with the old version. This greatly increases the learning cost for developers.

To mitigate this, researchers have proposed several automated approaches to repair incompatible APIs for Android apps. Fazzini et al. [44] introduced AppEvolve to update incompatible APIs based on examples of how other developers evolved their apps for the same changes. Similarly, Lamothe et al. [77] proposed an approach called A3 for supporting API migration through patterns mined from source code examples via abstract syntax tree (AST). These approaches require the target code under repair written syntactically similar to the before- and after-update API change examples. This makes them hard to find applicable updates, as claimed by Thung et al. [152]. To this end, Haryono et al. proposed CocciEvolve [56] to only learn updates from a single after-update example. It eliminates the weakness of AppEvolve by normalizing both the after-update example and the target app code. Nevertheless, the serious issue of CocciEvolve is its inability to resolve all the values used as API arguments. These can be expressed in various complex forms, e.g., field access expressions, method invocations, and object creations. Its other drawback is

the poor readability of its updated app code results. Its authors further extend their work by proposing AndroEvolve [55], which addresses the limitations of CocciEvolve through the addition of data flow analysis and variable name denormalization.

Unfortunately, all the above approaches focus on repairing the source code of Android apps and hence cannot be applied to directly repair published Android apps. Indeed, as time goes by, many published Android apps on popular app markets such as Google Play, unless being timely fixed, will become obsolete, leading to poor user experiences in the mobile ecosystem. To cope with this, market maintainers could choose to remove those apps. However, this may not be a good business model as it may reduce the market's competitiveness, i.e., its competitors are providing more choices of apps for users to explore. If market maintainers do not remove those apps, certain apps in the market will not be able to be installed on users' devices or will crash after installation. It will also cause poor user experiences and can even harm the reputation of the market and the app developers per se.

As a supplement to existing repair approaches that attempt to help developers in developing higher-quality apps, we believe there is also a need to provide approaches for helping repair published apps (before they are installed on users' devices), at least in the time period before their developers explicitly update the apps. This paper proposes a novel, generic approach for repairing three types of compatibility issues – API, device and callback-induced problems – in published Android apps. Since it is relatively simple to transform Java source code to bytecode and vice versa, the approach targeting published Android apps, with small change, could also be applied to repair apps at the source code level (but not the other way around). Nevertheless, we argue that source code- and bytecode-based repairing approaches are not mutually exclusive. They can co-exist and complement each other. Indeed, approaches targeting published app repair could be leveraged to achieve emergent repair, while source code-based approaches can come in later to gradually fix the issues. For example, market maintainers could leverage published app repairs to ensure the compatibility

of their hosted apps (with developers' permission) at app uploading or downloading time. This is extremely useful, especially for legacy apps that are less frequently (will no longer be) maintained by their developers. Correspondingly, end-users are provided options to use apps that could not be running initially on their devices.

In this work, we present a prototype tool, *RepairDroid* , which leverages pre-defined patch templates to instrument Android apps so as to fix compatibility issues. The templates are written by dedicated experts (i.e., app developers do not need to write patches for their apps) based on a structural model that is both descriptive and generic, i.e., a given patch should be applicable to all Android apps. The instrumentation process involves control-flow and data-flow analysis to locate and repair compatibility issues. Experimental results on thousands of real-world Android apps show that *RepairDroid* is effective in automatically repair various types of compatibility issues. In this research we make the following key contributions:

- We have designed a novel app patch description language and demonstrated that it is generic enough to be used to create fix templates for various compatibility issues. The genericity is achieved by allowing users to directly leverage the simple but well-defined Jimple grammar (i.e., a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java/Android bytecode) to describe the patches.

- We have designed and implemented a prototype tool *RepairDroid* , which follows given fix templates to automatically repair published real-world Android apps.

- We have evaluated our approach against 1,000 real-world Android apps. Experimental results show that our approach is effective in repairing Android apps, outperforms the state-of-the-art and achieves 85.34% of successful repairing rate.

Open source. The source code and datasets are all made publicly available in our

artifact package [2].

5.2 Motivation

The heavy fragmentation of the Android ecosystem has induced many types of compatibility issues in Android apps. Our research community has spent lots of effort on disclosing such issues, including at least the following three types:

OS-induced compatibility issues. This is one of the most common types of compatibility issues, where issues are caused by the evolution of the Android framework. During framework evolution, new APIs are regularly added to the framework, while existing APIs are also regularly deprecated and removed. In some rare cases, existing APIs may also be semantically changed, despite keeping the signature of the APIs unchanged. The example given in Listing 5.1 shows a deprecated API issue.

```
+ if (android.os.Build.VERSION.SDK_INT >= 28) {
+   for (Network nw: cm.getAllNetworks()) {
+     NetworkCapabilities nc = cm.getNetworkCapabilities(nw);
+     if (nc != null && nc.hasTransport(NetworkCapabilities.TRANSPORT_WIFI
    ))
+       return true;
+   }
+   return false;
+ } else {
    return an.getType() == ConnectivityManager.TYPE_WIFI;
+ }
```

Listing 5.1: An example of an OS-induced compatibility issue. API *getType* is deprecated at SDK level 28. On devices running SDK versions larger than 28, it is recommended to use API *hasTransport* instead.

Device-specific compatibility issues. These compatibility issues are associated with specific devices running customized Android systems. The problematic apps will only crash on certain devices while behaving normally on others, despite all the devices running the same Android framework version. Listing 5.2 presents such an example that was initially reported by Wei et al. [165]. The API *setRecordingHint* depends on a conditional statement that checks the device identifier against "Nexus 4". Only the condition to be true,

i.e., the corresponding app is indeed running on “Nexus 4”, the API will be executed.

```
   Camera mCamera = Camera.open();
   Camera.Parameters params =  mCamera.getParameters();
   ......
+ if (android.os.Build.MODEL.equals("Nexus 4")) {
+  params.setRecordingHint(true);
+ }
   ......
   mCamera.setParameters(params);
   mCamera.startPreview();
```

Listing 5.2: Patch for Camera Preview Frame Rate Issue on Nexus 4, excerpted from [165].

Inter-callback compatibility issues. This type of compatibility issue is caused by the changes to Android system callbacks (also known as lifecycle methods). Such system callback methods are pre-defined by the Android system and will be directly executed when certain conditions are satisfied. Listing 5.3 illustrates such an example that was initially reported by Huang et al. [63]. The *onAttach(Context)* callback method is only introduced from API level 23. If this code is running on smartphones with earlier API levels, this callback method will not be executed. Subsequently, the *mActivity* field will not be initialized, and its usage will likely throw *NullPointerException*s.

```
 public void onAttach(Context context) {
   super.onAttach(context);
-  mActivity = (BrowserActivity) context;
-  ......
+  attachActivity((BrowserActivity) context);
 }
+ public void onAttach(Activity activity) {
+  super.onAttach(activity);
+  if (Build.VERSION.SDK_INT < 23) {
+    attachActivity((BrowserActivity) activity);
+  }
+ }
+ private void attachActivity(BrowserActivity activity) {
+  mActivity = activity;
+  ......
+ }
```

Listing 5.3: The Patch for WordPress issue 6906. The compatibility issue is caused by the fact that the callback method *onAttach(Context)* is not yet available before API level 23, excerpted from [63].

All of these compatibility issues are equally critical to mobile apps as all of them will cause apps to crash, leading to poor user experiences. Compatibility issue repairing ap-

proaches should aim to fix all of them. However, current state-of-the-art tools only focus on repairing API-induced compatibility issues. Automatically fixing other types of compatibility issues, such as device or callback related ones, has not yet been addressed.

As summarized in Table 5.1, existing approaches also come with many limitations. For example, CocciEvolve only attempts to fix incompatible APIs within a single method. Their follow-up work AndroEvolve fixes this limitation by additionally introducing data-flow analysis into the fixing process. Compatibility issues with respect to (1) out-of-file variables and (2) multi-invocations in a single line and (3) compatibility issues in released Android apps cannot be resolved by any of the state of the art.

Table 5.1: Problematic Android compatibility issues addressed by state-of-the-art tools.

Feature	**A3**	**AppEvolve**	**CocciEvolve**	**AndroEvolve**
Out-of-method (within file) variables	✓	✗	✗	✓
Out-of-file variables	✗	✗	✗	✗
1-to-n replacement	✗	✓	✗	✗
Multi-invocations in a single line	✗	✗	✗	✗
Fix in published Android apps	✗	✗	✗	✗

5.3 Our approach: *RepairDroid*

We introduce *RepairDroid*, a template-based repair approach, to automatically repair three kinds of compatibility issues in Android apps. Figure 5.1 presents an overview of *RepairDroid*. It has three key modules: (1) Template Build Module (TBM), (2) Bug Location Module (BLM), and (3) Bug Repair Module (BRM). Below we detail these three modules.

5.3.1 TBM: Template Build Module

This first module of *RepairDroid* aims at preparing a set of semantic templates for subsequent modules to repair published Android apps. The main contribution of this module is a generic language for describing app patch templates. Developers should be able to

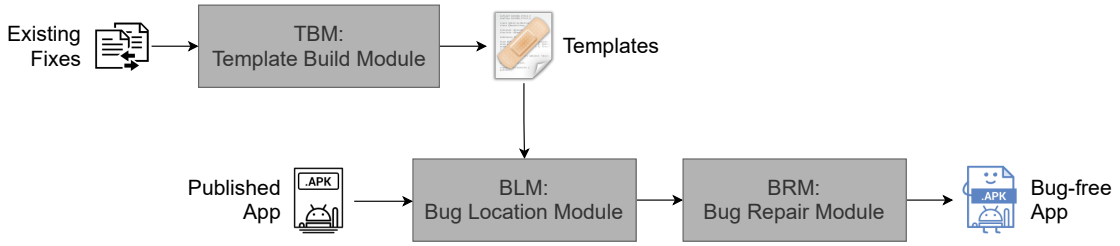


Figure 5.1: The working process of *RepairDroid*.

easily leverage the language to create templates for fixing compatibility issues, based on knowledge learned from the official Android documentation, online question and answer websites such as StackOverflow, or existing fixing samples mined from the evolution of real-world Android apps. The grammar of the language is straightforward. As shown in Figure 5.2, it contains three blocks, namely variable declaration block, issue location block, and patch block. We now detail these three blocks, respectively.

Variable Declaration Block. The first block provides a means for developers to specify all the variables involved in the template. At the moment, the language supports three types of variables.

- **Variables to be directly reused from the original context.** These variables appear in the original code statements (used for locating problems) and hence could be directly reused in the new code.

- **Variables to be searched from the original context.** These variables do not appear in the original code statements that are defined to locate the problems but do exist in the app context. Hence, backward data-flow analysis is demanded to search the definition of the variables for direct reuses. These variables will be explicitly marked

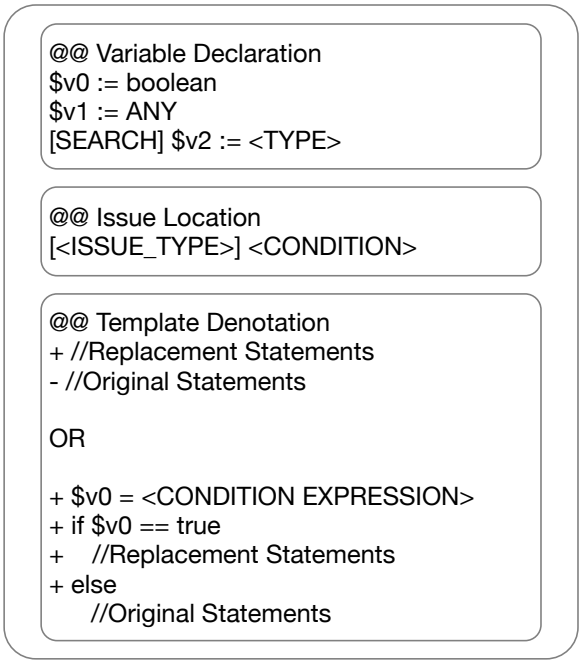


Figure 5.2: The language structure for creating patch templates for *RepairDroid*.

```
[OS] <android.net.NetworkInfo: int getType()> Build.VERSION.SDK_INT 28

[DEVICE] <Camera: Camera$Parameters getParameters()> Build.MODEL "Nexus
    4"

[CALLBACK] <Fragment: void onAttach(Context)> Build.VERSION.SDK_INT 23
```

Listing 5.4: Examples of issue location statements for creating templates to fix the compatibility issues listed in the three motivating examples (i.e., Listings 1-3), respectively

by keyword [SEARCH], as demonstrated in Figure 5.2.

- **Variables to be created.** These variables are totally new to the app and hence need to be defined before applying the template.

Issue Location Block. The second block is provided for developers to specify how the compatibility issues can be programmatically located. Developers first specify the issue type (in square brackets) to be fixed to simplify the location process. This is because different types of compatibility issues require different strategies to locate them. As inspired by the three types of compatibility issues summarized in Section 5.2, the language currently

provides three issue types (i.e., [OS], [DEVICE], and [CALLBACK], respectively for OS-induced, device-specific, and inter-callback compatibility issues) to guide *RepairDroid* for locating compatibility issues. More issue types could be added to the language if new types of compatibility issues are identified in the future.

Like the variable declaration block, the issue location block is also relatively easy to configure. In most cases, it only needs one line of statement to specify the issue. For example, the compatibility issues in the three motivating examples shown in Section 5.2 can be respectively specified by the issue location statements shown in Listing 5.4.

In addition to the targeted APIs, <CONDITION> also specifies the situation when the issue should not be considered, even if the targeted APIs or call methods are located. Taking Line 1 in Listing 5.1 as an example, if the invocation of the deprecated API *getType()* is already protected by an SDK version check (i.e., if (Build.VERSION.SDK_INT >= 28)), this target should be ignored as there should have no compatibility issue in such a case.

Template Denotation Block. The last module of the app patch language is used by developers to specify the actual template for fixing the located compatibility issues. The fix template is designed to be written in Jimple-like pseudocode. Jimple is the default intermediate representation of Soot [158], a well-known Java/Android static analysis framework. The reason why we choose Jimple to describe the template is that the instrumentation function of *RepairDroid* is implemented on top of Soot. It allows *RepairDroid* to quickly apply the template to fix compatibility issues in a deployed app[1].

As shown in Figure 5.2, the typical way to fix an OS-induced compatibility issue is to replace the original problematic statements with corrected new statements. In practice, we recommend the developers to always guard their newly introduced code through a con-

[1]Ideally, we would like to support Java as the language for describing the patch directly as it would be more convenient for patch writers. However, this will make the patch parsing step difficult to achieve as essentially it asks for a Java compiler to interpret the (random) Java code. Jimple is a simplified Java representation that could be an ideal trade-off solution, i.e., not very difficult to understand and write but can be interpreted programmatically in practice (thanks to Soot).

ditional check. When the check returns true, the replacement statements will be invoked. Otherwise, the original statements will be executed and hence the original behaviors are kept.

```
1  [Method] <[android.app.Fragment]: void onAttach(android.content.Context)
       >
2  [Stmt]$r0 := @this: [android.app.Fragment]
3  [Stmt]$a0 := @parameter0: android.content.Context
4  [Stmt]specialinvoke $r0.<android.app.Fragment: void onAttach(android.
       content.Context)>($a0)
5  [CUT] ... when != [END of Method]
6  + $v0 = ([android.app.Activity]) $a0;
7  + virtualinvoke $r0.<[android.app.Fragment]: void attachActivity([
       android.app.Activity])>($v0);
8  + return
9
10 + [NEW Method][public] <[android.app.Fragment]: void onAttach(android.
       app.Activity)>
11 + $r1 := @this: [android.app.Fragment]
12 + $a1 := @parameter0: android.app.Activity
13 + specialinvoke $r1.<android.app.Fragment: void onAttach(android.app.
       Activity)>($a1)
14 + $i1 = <android.os.Build$VERSION: int SDK_INT>
15 + if $i1 >= 23 goto <label_1>
16 + $v1 = ([android.app.Activity]) $a1;
17 + virtualinvoke $r1.<[android.app.Fragment]: void attachActivity([
       android.app.Activity])>($v1);
18 + <label_1>
19 + return
20 + [END of Method]
21
22 + [NEW Method][private] <[android.app.Fragment]: void attachActivity([
       android.app.Activity])>
23 + $r2 := @this: [android.app.Fragment]
24 + $a2 := @parameter0: [android.app.Activity]
25 [PASTE]
26 + return
27 + [END of Method]
```

Listing 5.5: The template denotation block of the inter-callback example.

Listing 5.5 shows an example Template Denotation block for the inter-callback example shown in Listing 5.3. Based on the example in Listing 5.5, we further introduce several keywords, such as [CUT], [PASTE], to respectively represent cut and paste operations, with which they can batch process statements within the set range, as shown at Line 5 and Line 25. Inspired by SmPL [78], on the basis of ensuring the universality of the language,

the templates for these three issues have designed some special symbols, including the use of some reserved words.

$$[CUT]...when! = [End\ of\ Method]$$

The "..." operator in SmPL represents an arbitrary sequence, i.e., any sequence of statements over any control flow path, which is also used in our language. For example, the usage of *when != [End of Method]* at Line 5 means that there should be no occurrences of *[End of Method]* in the matched control-flow path, that is, the matching process continues to the end of the method. Although there are no complicated usages in the repair of the three types of compatibility issues introduced above, our intention for this app patch language design is to ensure the future scalability of *RepairDroid*. To this end, *[End of Method]* can be replaced with other statements (e.g., tag statements such as <*label_1*>) to ensure that *RepairDroid* can be guided to accurately collect statements.

In real-world Android projects, the class inheritance feature has been frequently leveraged. In order to improve the versatility of *RepairDroid*, we use ["Superclass Name"] to specify the superclass of the class to be searched. For example, [android.app.Fragment] means *RepairDroid* needs to search such a class that extends the superclass named "android.app.Fragment" before performing subsequent work. If the superclass does not need to be restricted, [DECLARING_CLASS], a reserved keyword, can be directly used to indicate the class to which the current method belongs. *RepairDroid* will replace it with the actual value in the app when running on the BRM module.

5.3.2 BLM: Bug Location Module

The second module of *RepairDroid* takes as input the Android APK to be analyzed and the semantic templates generated by the first module and outputs the locations (at the statement level) indicating where the templates should be applied. Specifically, this module takes the following three steps to achieve its purpose: *template parsing*, *app pre-processing*, and *bug*

localization. We now detail these steps, respectively.

Template parsing. As a prerequisite to the following steps, *RepairDroid* first reads and parses the semantic templates generated by the TBM module. The parsing process utilizes the principle of Finite State Machine to parse the input semantic templates. After the parsing step, each semantic template is stored as a structured object that is readily available for further references.

App pre-processing. This step first transforms the bytecode into an intermediate representation code called Jimple, as it is non-trivial to directly analyze the Dalvik bytecode of Android apps. Jimple is the default intermediate representation format of Soot, a Java/Android app static analysis and optimization framework. In this work, *RepairDroid* leverages Soot to achieve the code transformation and the following-up static analysis of Android apps.

Bug localization. This step traverses the Jimple IRs of the target app and detects the locations that need to be patched according to the specified conditions parsed from the semantic templates above. *RepairDroid* automatically identifies the bug locations (often at the Jimple statement level) from the app code by traversing each method in each class. Subsequently, the located bug statements, along with their belonging methods, will be regarded as a potential bug candidate.

Then for each identified candidate, *RepairDroid* goes one step further to check if it satisfies certain conditions, following what will be also specified in the semantic template. If so, the candidate will be regarded as a true bug and hence will be propagated to the next module for automated repairing. For compatibility issues, the bug locations will often be an API invocation statement (e.g., because the API is no longer available in the latest Android devices or in certain customized Android versions such as Samsung phones). The conditions could be a framework version check or a device manufacturer check. Taking Listing 5.6 as a simple example, API *abandonAudioFocus* is deprecated in the Android

framework version M and hence is only recommended to be invoked on devices running lower versions than M. However, this method call should not be detected as containing a compatibility issue because the problem has already been protected (i.e., fixed). The conditional check (i.e., against *Build.VERSION_CODES.M*) should have been clearly specified in the semantic template of API *abandonAudioFocus*. To resolve this issue, after locating candidate bugs, *RepairDroid* goes one step deeper to perform an inter-procedural backward control-flow analysis to check if their associated conditions are presented. Only if the incompatible API calls are not already protected, *RepairDroid* will attempt to repair the corresponding issues.

We use the example shown in Listing 5.6 to illustrate this backward analysis flow. The deprecated API *abandonAudioFocus* and its replacement API *abandonAudioFocusRequest* are called in two separate methods, i.e., *abandonAudioFocus()* (i.e., Line 4) and *abandonAudioFocusRequest()* (i.e., Line 1), and the condition check statement is located in the method *dispose()* (i.e., Line 7). When locating an OS-induced issue, *RepairDroid* can easily pinpoint the deprecated API *abandonAudioFocus* (i.e., Line 5) and its declaring method, *abandonAudioFocus()*. However, there is no conditional check statement for the SDK version in the *abandonAudioFocus()* method. As a result, we need to locate method *dispose()* that calls the method *abandonAudioFocus()* through the backward analysis flow to determine that the current case does not contain an incompatible issue.

```
1  public void abandonAudioFocusRequest() {
2    audioManager.abandonAudioFocusRequest(request);
3  }
4  public void abandonAudioFocus() {
5    audioManager.abandonAudioFocus(this);
6  }
7  public void dispose(){
8    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M){
9      abandonAudioFocusRequest();
10   } else {
11     abandonAudioFocus();
12   }
13 }
```

Listing 5.6: Code snippets showing the demand for conducting inter-procedural backward flow analysis.

5.3.3 BRM: Bug Repair Module

The last module of *RepairDroid* is used to repair the located app compatibility bugs by directly updating the code snippets (hereinafter referred to as the *Target*) located by the BLM module, i.e., $Target \overset{Template}{\rightarrow} Target'$. The *Template* represents the list of statements used to update the *Target* that has been identified as containing bugs. Subsequently, $Target'$ represents the list of statements having the identified bugs repaired.

Algorithm 1 summarizes the general repairing process implemented in *RepairDroid*. *RepairDroid* first needs to ensure that all the variable keywords[2] involved in the semantic template are available in the code context. It transforms the variable keywords defined as strings in the template to Java objects at the code level. As demonstrated in Lines 2-12, for each variable keyword, *RepairDroid* first checks if its corresponding variable is directly available in the *Target* code. If so, it will be directly reused. If a given variable keyword is marked by keyword [SEARCH], we then consider the variable keyword needs to be searched in the *Target*'s context (the variable should have been defined in the app but has not been leveraged by the *Target* code). Finally, if the variable keyword is defined as a new variable, it will be directly initiated. Taking the code snippet shown in Listing 5.1 as an example, the variable *cm* (at Line 2) is such a case that needs to be searched backward, and *nw* (at Line 2) and *nc* (at Line 3) are variables that need to be created based on the searched variable *cm*.

When there are variables that do not exist in the original app code statements, *RepairDroid* leverages the *searchLocal* function to conduct backward control-flow analysis so as to select existing variables from the code context. The search process follows the following rules. These are summarized based on our manual observations among various fixes that happened in real-world Android apps. (1) Search backward from the position of *Target*

[2]To avoid confusion, we use variable keywords to describe the variables defined as strings in the template. The term variable by itself is kept for referring to Java objects.

for the variable in the located buggy statement's declaring method. The first seen variable (matched via type) will be considered. (2) If it is not possible to identify the variable in the current method, *RepairDroid* then traverses all the fields declared in the class to which the previously searched method belongs. Next, (3) if it still fails to locate the variable, *RepairDroid* will resort to the whole class, including its inner classes, to search for the variable. Finally, suppose it is still impossible to locate an existing variable after exploring all the aforementioned rules, *RepairDroid* will terminate the repair process and regard the fix as a failure case.

After preparing all the required variables, the next step is to update the *Target* code following the Jimple statements defined in the template. Since the Jimple statements are provided as strings, *RepairDroid* takes additional step to automatically transform it to code snippets (via the BUILD-JIMPLE-STMT function). Recall that the Jimple statements written in the template may contain certain placeholder keywords such as ["Superclass Name"], [CUT], and [PASTE]. When transforming the statements, *RepairDroid* needs to replace them with concrete values.

For example, as shown in Line 1 of Listing 5.5, *RepairDroid* first collects the actual name of the class inheriting the superclass *android.app.Fragment*, which contains incompatible issues. Then, it replaces [android.app.Fragment] with its real value, e.g., *org.wordpress. android.ui.themes.ThemeBrowserFragment* in the app of Listing 5.3, which extends the *android.app.Fragment* class, at runtime. Furthermore, some of the searched variables e.g. those returned by Rule (3)), may not be directly accessible in the method under repairing. To this end, *RepairDroid* goes one step further to introduce glue code to change the visibility of the search variables so that they can be freely used to repair the buggy statements.

Algorithm 1: The repair algorithm of BRM.

Input: $Target$: the located buggy statements.
$Template$: the classes/methods/statements that should be inserted.

```
1  Var2Local = new HashMap;
2  VariableKeywords = getVariableKeywords(Template);
3  Body = Target.getDeclaringMethodBody();
4  for var ∈ VariableKeywords do
5      if var shows in Target then
6          Local = getLocal(Target,var);
7      else if var needs to be searched then
8          Local = searchLocal(Body, var);
9      else
10         Local = new Local(Body, var, Var2Local);
11     Var2Local.add(var, Local);
12 end
13 for Item ∈ Template_before do
14     NewStmt = BUILD-JIMPLE-STMT(Item, Var2Local);
15     insertBefore(NewStmt, Target.get[0]);
16 end
17 for Item ∈ Template_after do
18     NewStmt = BUILD-JIMPLE-STMT(Item, Var2Local);
19     insertAfter(NewStmt, Target.get[Target.size() − 1]);
20 end
```

5.3.4 Implementation

RepairDroid is implemented on top of Soot and the repair process is done at the Jimple code level. Thanks to Soot, the repaired Jimple code is further transformed back to a new Android app, which could be directly installed and used by users. Although our approach is proposed to repair Android apps directly at the bytecode level, app developers who want to repair their apps at the source code level could also benefit from our approach. By reverse-engineering the repaired Android apps, developers can get fixed source code, which could then be directly ported to the source code project to achieve source code repairs.

5.4 Evaluation

The goal of this work is to automatically repair compatibility issues in published Android apps. To determine if this objective has been achieved, we look to answer the following four key research questions:

RQ1: How generic is our proposed patch description language?

RQ2: How well does *RepairDroid* perform compared with existing tools?

RQ3: How effective is *RepairDroid* in automatically locating and repairing incompatibility issues in real-world Android apps?

RQ4: What is the time performance of *RepairDroid* in repairing published Android apps?

5.4.1 RQ1: Genericity

In the first research question, we investigate the genericity of our proposed patch description language, which is one of the core modules supporting the automated repair of Android apps. As discussed in Section 5.2, there are at least three types of compatibility issues suffered by real-world Android apps. The issues are significantly different from one to another. Even within the same type, the compatibility issues could also be remarkably different. Recall that we aim to design the patch description language to be as generic as possible so that it can be leveraged to describe templates for all the kinds of compatibility issues Android apps may encounter. To this end, we resort to evaluating the language's genericity by directly applying the language to create templates for all the compatibility issues explicitly mentioned in the three articles in which the aforementioned three types of compatibility issues (cf. Section 5.2) are introduced, respectively.

By manually summarizing the examples detected by the previously mentioned three approaches (i.e., CiD, Pivot, and CIDER, respectively), we eventually decided to create templates for three OS-induced compatibility issues, seven device-specific compatibility issues, and six inter-callback compatibility issues. In the work of CiD, the authors listed seven issues, of which only three have been confirmed and fixed by the developers. Hence we only take these three issues into consideration. Furthermore, the authors of CiD have further presented another work, CDA [93], that reveals 19 more issues. As a supplement,

we decided to also consider them to create OS-induced compatibility issues. However, for five of them, we cannot find intuitive replacements and hence these were excluded before the experiment. These five issues are associated with the Apache HttpComponents project, for which the whole project has now been deprecated by the Android framework. To address them hence requires fundamental code changes – removal of all the usage of Apache HttpComponents – to repair their related app compatibility issues. Such a change is far too difficult to be done automatically. We thus eliminated these five compatibility issues from this experiment. As a result, 17 OS-induced compatibility issues are considered, as enumerated in the second column in Table 5.2. In the work of Pivot, the authors present ten problematic apps, among which only seven compatibility issues are eventually fixed. These seven issues correspond to three distinct APIs. In the work of CIDER, the authors have reported nine issues, among which only six issues (correspond to four distinct methods) are eventually fixed. To ensure that we can find repair examples that provide specimens demonstrating how these compatibility issues can be fixed and thereby how to describe the templates, we decided to only focus on the three device-specific and four inter-callback compatibility issues that have been fixed or at least confirmed by the developers. The detailed selected issues are also listed in the second column in Table 5.2.

To evaluate the genericity of *RepairDroid* 's patch description language, for each of the selected compatibility issues in Table 5.2, the authors manually analysed each issue and its correct fixes and created templates following the rules defined by the language presented earlier. As shown in Table 5.2, for the 24 considered compatibility issues across three different types, **we successfully created 22 templates for these 24 app compatibility issues**. We will evaluate the correctness of these templates when answering the third research question.

The remaining two issues we could not fix using our *RepairDroid* patch description language. That is because these two issues have not been provided with clear replacement

Table 5.2: Experimental results demonstrating the genericity of the patch description language.

Paper	Issue	If success	Total
CiD [91] CDA [93]	<Resources: Drawable getDrawable(...)>	✓	**15/17**
	<Notification: void setLatestEventInfo(...)>	✓	
	<View: void setBackgroundDrawable(...)>	✓	
	<Intent: ClipData getClipData()>	✓	
	<View: void setSystemUiVisibility(...)>	✓	
	<Notification.Builder: Notification.Builder setLocalOnly(...)>	✗	
	<Notification: void <init>(...)>	✓	
	<NetworkInfo: int getType()>	✓	
	<Display: int getWidth()>	✓	
	<Display: int getHeight()>	✓	
	<Resources: int getColor(...)>	✓	
	<PopupWindow: void setWindowLayoutMode(...)>	✓	
	<Activity: void setProgress(...)>	✗	
	<ContentProviderClient: boolean release()>	✓	
	<AccessibilityServiceInfo: String getDescription()>	✓	
	<AccessibilityServiceInfo: boolean getCanRetrieveWindowContent()>	✓	
	<Html: String toHtml(...)>	✓	
Pivot [165]	<DatePickerDialog: DatePickerDialog DatePickerDialog(...)>	✓	**3/3**
	<View: int getSystemUiVisibility()>	✓	
	<Camera$Parameters: void setRecordingHint(...)>	✓	
CIDER [63]	<Fragment: void onAttach(...)>	✓	**4/4**
	<WebViewClient: boolean shouldOverrideUrlLoading(...)>	✓	
	<WebViewClient: onReceivedError(...)>	✓	
	<WebViewClient: onReceivedHttpError(...)>	✓	

information indicating how they can be avoided on the official Android documentation site. These failures, however, have no connection with the genericity of the language. Indeed, if we are provided with well-structured repair samples, we could still generate templates for these two issues. Overall, this experimental result shows that our patch description language is quite generic and should be capable of describing patterns for finding and fixing a wide range of compatibility issues encountered by Android apps.

Answer to RQ1:

RepairDroid 's patch description language is generic and should be capable of describing most of the compatibility issues available in the Android ecosystem.

5.4.2 RQ2: Comparison With State-of-the-art

To answer our second research question, we compare *RepairDroid* with all the state-of-the-art related works. To the best of our knowledge, as listed in Table 5.1, there are four tools (i.e., A3 [77], AppEvolve [44], CocciEvolve [56], AndroEvolve [55]) proposed by our fellow researchers used to automatically fix OS-induced compatibility issues in Android apps. However, A3 is mainly developed to mine migration patterns from code examples. Although it provides mechanisms to automatically apply the mined migration patterns to fix the problematic APIs, there is no guarantee that such attempts will be correct. We believe it is not fair to compare our approach with A3 and hence exclude A3 from the comparison.

Furthermore, because of certain limitations of AppEvolve, e.g., it cannot handle updates spanning multiple methods, CocciEvolve is proposed to complement AppEvolve. CocciEvolve is able to achieve better performance than AppEvolve on 112 target problems. The authors of CocciEvolve further propose AndroEvolve that extends CocciEvolve to achieve even better performance in automatically repairing incompatible APIs. Therefore, in this work, we compare our approach with AndroEvolve, the most relevant and advanced approach closest to ours.

In order to experimentally compare the performance of *RepairDroid* against AndroEvolve, we need to ensure that these two tools are launched to repair the same set of compatibility issues. We hence made effort to create compatibility patch templates for all of the issues targeted by AndroEvolve. Specifically, AndroEvolve was evaluated against 20 problematic APIs, as shown in the first column in Table 5.3. Following *RepairDroid* 's description language, we were able to create templates for all the 20 APIs. This also further demonstrates the genericity of *RepairDroid* 's patch description language, complimenting

our RQ1 answer above.

We then launch *RepairDroid* and AndroEvolve to repair Android apps that contain the aforementioned 20 compatibility issues. We build the corresponding projects used in the evaluation of AndroEvolve into apps and conduct experiments based on them, as *RepairDroid* requires published Android apps to check if *RepairDroid* can repair the corresponding issues. Table 5.3 summarizes the experimental results. ***RepairDroid* is able to successfully repair all the APIs**, including all of the ones that cannot be handled by AndroEvolve. As explicitly acknowledged by Hartono et al. [55], AndroEvolve cannot handle the updates of a single API into multiple APIs. Therefore, AndroEvolve fails to repair *getAllNetworkInfo()* API as its fix requires to access two APIs, i.e., *getAllNetworks()* and *getNetworkInfo()*. Moreover, AndroEvolve cannot deal with the case when multiple API invocations are written in a single line of code. It is indeed non-trivial to resolve this challenge as it may involve complicated operations in that line of code. However, this challenge will not be an issue for *RepairDroid*. Indeed, *RepairDroid* repairs Android apps at the Jimple code level for which the multiple invocations are separated into different lines. These experimental results show that *RepairDroid* goes beyond the state-of-the-art to repair compatibility issues in Android apps.

Answer to RQ2:

RepairDroid outperforms the state-of-the-art tools by achieving better performance in automatically repairing compatibility issues in Android apps.

5.4.3 RQ3: Effectiveness of *RepairDroid*

Our third research question concerns the effectiveness of *RepairDroid* in automatically repairing compatibility issues in published Android apps. To answer this RQ we randomly selected 1,000 real-world Android apps from AndroZoo to form our test dataset. These 1,000 apps were originally collected from the official Google Play store and hence are all published apps (i.e., their source codes are not available anyway). Recall that, when

Table 5.3: Comparison results between *RepairDroid* and the state-of-the-art AndroEvolve approach.

API	AndroEvolve	RepairDroid
<AccessibilityNodeInfo: void addAction(...)>	✗	✓
<ConnectivityManager: NetworkInfo[] getAllNetworkInfo()>	✗	✓
<TimePicker: Integer getCurrentHour()>	✓	✓
<TimePicker: Integer getCurrentMinute()>	✓	✓
<TimePicker: void setCurrentHour(...)>	✓	✓
<TimePicker: void setCurrentMinute(...)>	✓	✓
<TextView: void setTextAppearance(...)>	✓	✓
<LocationManager: boolean addGpsStatusListener(...)>	✓	✓
<Html: Spanned fromHtml(...)>	✓	✓
<ContentProviderClient: boolean release()>	✓	✓
<LocationManager: boolean addGpsStatusListener(...)>	✓	✓
<WebViewClient: boolean shouldOverrideUrlLoading(...)>	✗*	✓
<View: boolean startDrag(...)>	✓	✓
<AudioManager: int abandonAudioFocus(...)>	✗	✓
<TelephonyManager: String getDeviceId()>	✓	✓
<AudioManager: int requestAudioFocus(...)>	✓	✓
<Canvas: int saveLayer(...)>	✓	✓
<MediaPlayer: void setAudioStreamType(int)>	✗	✓
<Vibrator: void vibrate(long)>	✓	✓
<Vibrator: void vibrate(long[],int)>	✓	✓

* No examples and scripts provided.

answering RQ1 and RQ2, we have created templates for 42 distinct compatibility issues. We used all of these 42 templates when applying *RepairDroid* to try and detect and repair compatibility issues in the randomly selected apps. Note that at this stage, we do not know yet whether these apps contain true compatibility issues or not.

Table 5.4 summarizes the experimental results. 714 apps contain potential OS-induced compatibility issues (i.e., 8,519 in total[3]), and 492 apps suffer from potential device-specific issues (i.e., 3,086 in total). Only seven apps contain potential inter-callback compatibility issues (i.e., 31 in total). In this experiment, we consider an app that contains inter-callback compatibility issues only if the unsupported callback methods are explicitly overridden by developers. Among all the identified issues, *RepairDroid* locates that 5,932 OS-induced,

[3]Issues lie in Android framework code are ignored.

3,042 device-specific, and 2 inter-callback compatibility issues are true issues for which they are not already protected. For each of the located issues, *RepairDroid* then applies its corresponding template to perform the automated repair. Eventually, 4,616 OS-induced, 3,042 device-specific, and 2 inter-callback compatibility issues can be successfully fixed, giving a success rate at 77.82%, 100%, and 100%, respectively. To validate the fixes, we randomly sample 20 apps and leverage Soot's grammar checker to check if their updated code is grammatically correct. Then, we evaluate the repaired app through (1) manually comparing the repaired code with the original buggy code and (2) actually executing the repaired apps (as well as their original counterparts) to verify the fixes of the corresponding compatibility issues. The repaired code of the 20 sampled apps has been manually verified and confirmed to be correct, and all the repaired apps can also be normally installed on Android devices. **Overall, *RepairDroid* is able to achieve an 85.34% of success rate when repairing 1,000 randomly selected Android apps.** Figure 5.3 plots the distribution of the number of detected and the number of successfully fixed compatibility issues, i.e., the median and mean numbers are 10, 12.77, and 9, 11.12, respectively. The fact that the majority of located issues can be automatically repaired demonstrates the effectiveness of our approach.

The failure cases are mainly caused by the variable search module, for which *RepairDroid* fails to pinpoint the required variables based on the variable keywords leveraged in the template. Taking the code snippet displayed in Listing 5.1 again as an example, in our experiment, a total of 760 compatibility issues related to API *getType* are located. However, for around 25% at the moment, *RepairDroid* currently fails to search and locate the actual variable for keyword *cm*. Furthermore, due to limitations of Soot, *RepairDroid* also fails to repair a number of issues. Moreover, there might be multiple ways to fix an issue. We respectively provide a general repair template targeting each issue, which may not fully meet all cases' requirements. As of our future work, we commit to continuously improve our ap-

proach to increase its success rate in automatically fixing compatibility issues in published Android apps.

Table 5.4: Performance achieved by *RepairDroid* for repairing 1,000 randomly selected Android apps.

Issue Type	# Apps	# Potential Issues	# Located Issues (No Protection)	# Fixed Issues
OS-induced	714	8,519	5,932	4,616
Device-specific	492	3,086	3,042	3,042
Inter-callback	7	31	2	2
Total	725*	11,636	8,976	7,660 (85.34%)

* One app may suffer from multiple types of issues.

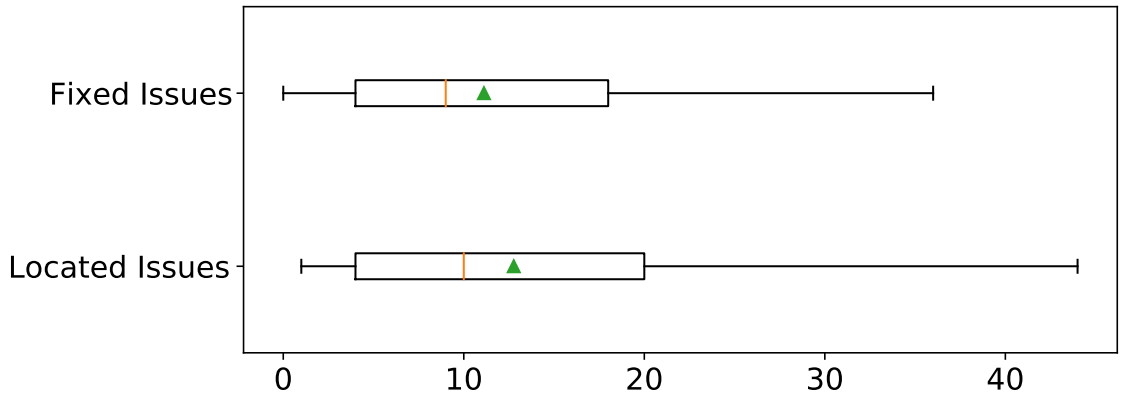


Figure 5.3: Distribution of the located and successfully fixed compatibility issues in each of the selected apps.

Answer to RQ3:

With an overall 85.34% of success rate, the experimental results show that *RepairDroid* is effective in automatically repairing real-world Android apps.

5.4.4 RQ4: Time Performance of *RepairDroid*

In our last research question, we investigate the time performance of *RepairDroid* when applied to repair real-world published Android apps. Taking the same 1,000 apps as lever-

aged for answering RQ3 as input, Figure 5.4 presents the distribution of the execution time that *RepairDroid* spends for analyzing an app. The distribution is plotted with respect to the number of located compatibility issues identified in an app and the DEX size of each app, respectively. As the number of located issues per-app increases, the time spent to repair the app also increases. Nevertheless, the increase seems to be gradual. Indeed, the Pearson correlation coefficient confirms that there is only a weak correlation ($r = 0.34, p - value < 0.001$) between them. When DEX size is concerned, we cannot observe direct connections between the DEX size of the app and the time that *RepairDroid* spends to repair the app. Pearson correlation analysis also confirms our observation that there is literally no correlation ($r = -0.07, p - value = 0.05$) between them.

The results suggest that **the time performance of *RepairDroid* is quite stable in analyzing a real-world Android apps**, no matter how large the code size is or how many compatibility issues it suffers from. This evidence further suggests that our approach is suitable to be applied to repair large-scale Android apps.

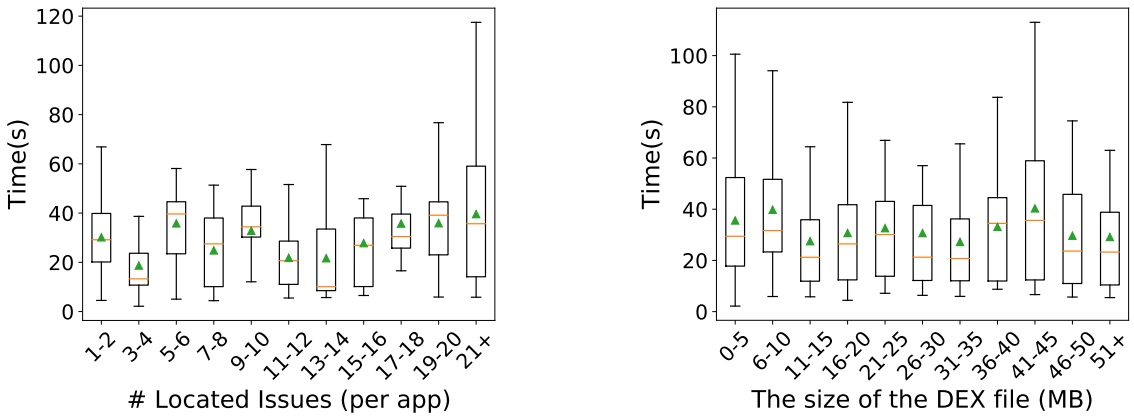


Figure 5.4: Distribution of the execution time with respect to the number of issues contained per app and its DEX size, respectively.

Answer to RQ4:

The time spent by *RepairDroid* to repair a real-world Android app is stable, with no impact by the app's code size and only a slight impact by the number of compatibility issues that needed to be fixed.

5.5 Discussion

5.5.1 Limitations

While *RepairDroid* aims for precise and sound analysis and repair, it does share some inherent limitations with most other static analysis tools [90]. *RepairDroid* is oblivious to reflective calls, native code, multi-threading features, which may impact the static analysis results and hence lead to inaccurate locations of compatibility issues. As of our future, we plan to leverage the DroidRA [88, 89, 148] and JuCify [142] tool to mitigate the impact of reflective calls and native code, respectively. Currently *RepairDroid* is only aware of the most common expression types defined in Soot. In some rare cases, when the reserved keywords such as [CUT] and [PASTE] involve unusual expressions, *RepairDroid* may not be able to recognize them and hence will not generate executable statements, resulting in incorrect repairs. However, it is relatively easy to extend *RepairDroid* for including more of Soot's expression types. This limitation could be mitigated in practice. Furthermore, we have only evaluated the correctness of device-specific compatibility issue repairings through manual confirmation because we cannot find the relevant devices for testing. Such manual processes are, however, known to be error-prone. To mitigate the threat, we have cross-validated the results. We have also released our tool and dataset for public reference.

5.5.2 Mining Fix patterns from existing Android apps

RepairDroid is limited by the set of templates prepared for analysis. The more templates created and included, the more compatibility issues will likely be automatically fixed. However, it might be non-trivial to create templates for some issues as it could require compli-

cated background knowledge to understand the issues and their corresponding fixes. This burden could be significantly mitigated if we can locate real-world code examples relevant to fix compatibility issues. Following the idea of Fazzini et al. [44], who propose to learn fix patterns from the evolution of open-source Android apps, we propose to mine such fix patterns from the evolution of published Android apps. We believe the evolution of published Android apps could provide much more knowledge than mining open-source apps since the majority of apps have only released their published versions (i.e., source code not available).

Table 5.5: The list of the top-5 API Pairs mined from real-world Android apps.

Incompatible API	**Replacement API**	**# Apps**
setBackgroundDrawable	setBackground	5,382
isScreenOn	isInteractive	2,129
getDrawable(int)	getDrawable(int,Theme)	1,543
setOnUtteranceCompletedListener	setOnUtteranceProgressListener	1,240
getColorStateList(int)	getColorStateList(int,Theme)	1,085

To this end, we randomly selected 20,000 apps from AndroZoo [10] and conducted a lightweight static analysis to locate API calls that are protected by SDK version checks (e.g., based on the following pattern: if (condition) $API_{incompatible}$ else $API_{replacement}$). To ensure that the located API calls are indeed relevant to fix compatibility issues, we further resort to the list of 228 API deprecated-and-replacement pairs disclosed by Li et al. [92]. We consider a potential fix pattern is located if the following conditions are satisfied: (1) the API calls are protected by SDK version checks, (2) the problematic API and its replacements are respectively presented in the two branches separated by the SDK version check. Among the 228 pairs, we are able to locate 34 practical fix patterns, for which the top-5 ones are illustrated in Table 5.5. Following the located fix patterns, we could indeed easily create templates for the involved problematic APIs. Those templates can subsequently help *RepairDroid* in automatically repairing the pinpointed issues in the corresponding

real-world apps. This preliminary study experimentally shows that our approach *RepairDroid* could indeed benefit from existing compatibility issue fixes conducted by developers in practice. This study is our initial attempt and is part of our ongoing efforts at creating more templates for repairing compatibility issues in published apps.

5.5.3 Supporting automated repair for both Java and Kotlin written Android apps

Since 2017, Google has promoted Kotlin as the official programming language for developing Android apps. In 2019, Google further declared that Android becomes 'Kotlin-first' (i.e., new API, libraries, documentation will target Kotlin first) and hence advised developers to develop new apps using Kotlin instead of Java. Since then, more and more Android apps have completely migrated from Java to Kotlin. However, to the best of our knowledge, all existing app repairing approaches only focus on Java written apps, letting a large number of Kotlin written apps untouched. Indeed, since Java and Kotlin are two different programming languages, code repairing approaches proposed for one language cannot be applied to the other. Analysts have to completely rewrite the code parser and repair module following the new languages' syntax. We hence argue that there is a need to support automated repair for both Java and Kotlin written apps.

Fortunately, no matter which language is leveraged, the published Android apps will be in Dalvik bytecode. Therefore, *RepairDroid* could be directly applied to repair both Java and Kotlin written apps. For example, *RepairDroid* tool can perform successful fixes on the DuckDuckGo-Kotlin [1] app.

5.6 Related work

Compatibility Analysis: Compatibility issues have been a key research topic in the Android community [164, 160, 145, 165, 63, 91, 105, 26]. To assist developers in exhaustive app testing, Wei et al. [164] empirically study the fragmentation-induced issues to char-

acterize the symptoms and root causes and propose a technique named FicFinder to detect such compatibility issues. After that, the authors [165] further present an API-device correlation extracting and learning approach named Pivot to help detect fragmentation-induced compatibility issues. Huang et al. [63] delve into the callback API evolution induced compatibility issues and provide a technique named CIDER, leveraging a graph-based model to detect two types of callback compatibility issues. Unfortunately, both Pivot and CIDER focus on detecting some types of incompatibility issues instead of repairing them, the motivation of our *RepairDroid*.

The exploration of compatibility issues caused by Android OS evolution is needed as apps are inseparably linked to the official Android APIs. Researchers have put a lot of effort into deprecated APIs [173, 92, 93, 57, 56, 44, 87], which could eventually lead to compatibility issues. Li et al. [92] build a prototype tool called CDA and apply it to different revisions of the Android framework to characterize deprecated Android APIs. Based on an extensive empirical study, He et al. [57] reveal that drastic API changes exist between neighboring Android versions. They have additionally developed a tool named IctApiFinder to detect incompatible API usages. Similarly, Li et al. [91] propose an approach named CiD to model the lifecycle of the Android APIs and flag the error usages capable of causing compatibility issues, the issues declared by which are also regarded as one of our motivations. Xia et al. [170] perform a large-scale study on the practice of handling OS-induced API compatibility issues and their solutions, and propose a tool, RAPID, to ascertain whether a compatibility issue has been addressed.

In non-Android communities, research on API deprecated is also ubiquitous [137, 60, 188, 22, 144, 162]. Zhou et al. [188] study API deprecation usage in open-source Java frameworks and libraries. They also propose a framework to detect deprecated API usages in source code examples on the Web. Brito et al. [22] perform a large-scale analysis of real-world Java systems and reveal that there is almost no significant effort to improve

deprecation messages. Some researchers concentrate on the impact of API deprecation [60, 137, 144, 33]. Hora et al. [60] report on an exploratory study that aims to observe API evolution and its impact on the Pharo ecosystem. Sawant et al. [144] extend the study on Java and investigate how many API clients update their dependencies to actively maintain their projects and count the number of affected projects by deprecation.

Program Repair: Program transformations and repairing have been widely researched [23, 67, 70, 78, 139, 114, 159, 34, 102, 104, 157, 74, 75, 156]. For instance, LASE [67, 114] is an example-based program repair tool by learning non-trivial data and its context from multiple editing examples and automatically searching for editing locations to apply customized editing to these locations. Liu et al. [98] present a technique based on the code edits performed by developers for automatically learning program transformations that are leveraged to repair program defects automatically [100, 101], while the patch validation of such program repair relies on the test cases of the targeted programs [99, 103, 132, 182]. Coccinelle [23, 78] is a C-based program matching and source-to-source transformation tool that has been employed for the automatic evolution of the Linux kernel. Coccinelle provides Semantic Patch Language (SmPL) to write its transformation rules. As a Java extension to Coccinelle, Coccinelle4J [70] is designed to apply for Java programs. Similar to Coccinelle, it uses semantic patches written in SmPL.

Recently, studies have begun to focus on the usage of automatically updating incompatible Android APIs. As one of the first tools to implement this goal, AppEvolve [44] using GitHub as the code base to perform API updates by learning examples before and after the update. Haryono et al. [56] improve AppEvolve by proposing CocciEvolve, which uses a single updated example to perform API updates and provides readable and configurable scripts in the form of semantic templates. They further broaden their study by proposing AndroEvolve [55], which addresses the defects of CocciEvolve with data flow analysis and variable name denormalization. Similar to their works, *RepairDroid* also rely on semantic

templates to automatically perform API updates. Lamothe et al. [77] leverage the basic *diff* in the version control system to learn API migration patterns, where they use the ASTs to match the API calls in the source code to the code examples. Unlike the above researches that directly act on the Java source code, our study concentrates on the low-level programming language, intending to modify Dex files directly and break the limitation that they can only take effect within the scope of a method or file.

5.7 Conclusion

We have proposed a novel prototype tool *RepairDroid* for automatically repairing compatibility issues in published Android apps. *RepairDroid* provides a patch description language for users to create fix templates for given compatibility issues. *RepairDroid* then applies these created templates directly to the Android app bytecode to repair the corresponding compatibility issues. Experimental results show that the patch description language is generic, being able to correctly describe 42 out of 44 issues, and the repair module is effective, being able to outperform the state-of-the-art approaches and achieve an overall 85.34% of success rate in repairing 1,000 real-word Android apps. *RepairDroid*'s execution time per app is also stable, making it suitable to be applied to conduct market-scale repairings.

Chapter 6

Conclusion and Future Work

6.1 Key Contributions

This Ph.D. project aimed to utilize big code analysis-based techniques for enhancing the quality of Android apps. Specifically, we outlined three key approaches to achieve this goal:

First, we empirically investigated the **impact of sample duplication on machine learning-based Android malware detection approaches**. We started by recognizing common sample duplication types in well-known and used Android malware datasets. We then took into account these sample duplication types to train distinctive machine-learning models to classify Android malware. We conducted our experiments on three common malware datasets. Our experimental results show that sample duplication does indeed impact the performance of machine-learning-based malware detection approaches. An in-depth exploration further revealed that this finding applied to not only in-the-lab experiments (i.e., 10-fold cross-validation) but also in-the-wild analyses (i.e., trained on one dataset and then tested on another). This finding also applies to experiments that were conducted using different machine learning algorithms, including both supervised and unsupervised learning approaches. We summarise the key contributions of this work as follows:

- We empirically investigated the impact of sample duplication on machine learning-

based Android malware detection approaches.

- We took into account these sample duplication types to train distinctive machine-learning models to classify Android malware.

- We argue that our fellow researchers and practitioners should always take sample duplication into consideration when performing machine learning-based (via either supervised or unsupervised learning) Android malware detections.

Second, we have proposed a **novel multi-dimensional context-aware collaborative filtering-based approach, APIMatchmaker**, for recommending API usages to Android app developers. **APIMatchmaker** initially leverages both pure code implementations and app topics to identify high-level features and subsequently locate similar projects as well as methods that are closest to the active project's method under editing. Then, it utilizes the encoding matrix and rating algorithm to obtain the output of recommended APIs and extracts the code snippets (as API usage samples) from the original APK files. Our experimental results on large-scale datasets demonstrate that **APIMatchmaker** is effective in learning and recommending API usages for Android developers and is able to outperform both state-of-the-art and our baseline approaches. We summarise the key contributions of this work as follows:

- We introduce to the community a new multidimensional context-aware collaborative filtering approach to better locate the most similar apps to support the recommendation.

- We design and implement a prototype tool, which takes as input a method under editing and outputs a list of APIs (and their usage samples) meeting the constraints of the SDK versions that could be leveraged to complete the implementation of the method.

Third, we have proposed a **novel prototype tool RepairDroid for automatically repairing compatibility issues in published Android apps**. RepairDroid provides a patch description language for users to create fix templates for given compatibility issues. RepairDroid then applies these created templates directly to the Android app bytecode to repair the corresponding compatibility issues. Experimental results show that the patch description language is generic, correctly describing 42 out of 44 issues, and the repair module is effective, outperforming the state-of-the-art approaches and achieving an overall 85.34% of success rate in repairing 1,000 real-world Android apps. RepairDroid's execution time per app is also stable, making it suitable to be applied to conduct market-scale repairings. In this work, we make the following key contributions:

- We have designed a novel app patch description language and demonstrated that it is generic enough to be used to create fix templates for various compatibility issues.
- We have designed and implemented a prototype tool, which follows given fix templates to repair published real-world Android apps automatically.

6.2 Limitations

Several limitations may have impacted the validity of the work presented in this thesis, and below we will highlight the key threats.

Exhaustiveness of the classification algorithms: For our first task of big code reliability analysis for malware detection approaches, the main threat to construct validity concerns the exhaustiveness of the classification algorithms we selected and the experimental setups. Although we have selected four well-known algorithms and both in-the-lab and in-the-wild experimental settings, which have also been frequently leveraged by other researchers to achieve similar purposes, they may not be entirely suitable for predicting Android malware [151]. Nonetheless, the four algorithms yielding more or less similar results

suggest that our findings are not specific to a particular learning algorithm. The key threat to internal validity concerns possible errors in the implementation of our experimental tools and scripts used to run the experiments and gather experimental results.

To reduce this threat, we have carefully reviewed the code and scripts of our toolchain to ensure that the implemented functions meet our expectations. We have further manually checked a random selection of experimental results to verify their accuracy.

Representativeness of the malware datasets: One threat to the external validity of our study concerns the representativeness of the malware datasets that we selected. Although we have included four common malware datasets from the literature, our results may still not be generalizable to other malware datasets. Nonetheless, the fact that our experimental findings are similar among the selected datasets shows that the external validity of our work is likely to be reasonable. Also, to avoid potential biases, we restrict the test dataset to contain unduplicated samples only when conducting the supervised learning experiments, which unfortunately may not reflect the real-world situation as it is likely to have duplicated samples in a real-world dataset.

Nevertheless, since this decision will not impact the capability of the classifier (which only relies on the training dataset) and the duplication rate in practice is not significant, such a decision should only bring limited threats to our experiments and hence could be neglected.

Authenticity of malware labels: Another limitation of this work is that the performance of our family clustering experiments is directly related to the authenticity of malware labels. These, unfortunately, may not be reliable, as often discussed by other researchers [65, 49, 147]. To mitigate this threat, we have directly leveraged the malware labels provided by the malware datasets, which have already been leveraged by various prior research projects.

Random selection of our training dataset: For our second task of API recommenda-

tions for Android app development, one of the main threats to external validity lies in the random selection of our training dataset, which may not be generic and representative of all the apps available in the ecosystem. We have attempted to mitigate this threat by selecting more than 10,000 real-world apps that have been all released over the official Google Play store. Additionally, we also evaluate the effectiveness of our approach by varying the size of the training dataset. The fact that there is no significant difference observed by doing that shows that this threat to the external validity of our study could be small.

App Obfuscation issues: At the moment, we do not take into account app obfuscation in this work, despite it has likely been applied to some of the apps in our training dataset. Nonetheless, since we are mainly interested in learning API usages, which would not be impacted by simple obfuscation strategies such as method renaming, our approach should remain valid and effective. Indeed, as revealed by Dong et al. [40], the majority of Android apps are only obfuscated via simple strategies. Advanced strategies such as altering the code implementation are rarely adopted by real-world Android apps.

Accuracy of App Descriptions: We leverage the app description on Google Play to represent the app topic. We consider these descriptions reliable because they are publicly advertised by the app owners. However, the app descriptions may not be well-written or representative of the actual app topic. Ideally, it would be great if automated approaches could be introduced to validate the app descriptions before applying them to our approach. This is nonetheless out of the scope of this work.

Reliability issues of static analysis: The implementation of *APIMatchmaker* relies on two static analysis frameworks, Soot [76] and JavaParser[1], for which their reliability issues could propagate to *APIMatchmaker* so as to threaten the validity of our approach. Nevertheless, both Soot and JavaParser are well-known frameworks that have already been shown practical and useful by many of our fellow researchers [15, 83, 36]. We hence

[1]https://javaparser.org

believe the threat brought by these two tools should not be significant.

Use of simulated scenarios: Another key limitation is that we employ simulated experimental scenarios for evaluation rather than user studies. Drawing on the related work [119], we mitigate the threat by forming large-scale datasets with apps randomly collected from AndroZoo [10] and adopting 10-fold cross-validation to reduce the impact of contingency. We also attempt to simulate different development scenarios (at an earlier or later stage of the development) and examine the recommended results of *APIMatchmaker* with the ground truth composed of the real APIs invoked by the method under editing. Nevertheless, we believe user studies are also necessary towards understanding if our approach has fulfilled the actual demand of Android app developers.

Native code, reflection and multi-threaded code issues: In our third research on compatibility issue repair, while *RepairDroid* aims for precise and sound analysis and repair, it does share some inherent limitations with most other static analysis tools [90]. *RepairDroid* is oblivious to reflective calls, native code, and multi-threading features, which may impact the static analysis results and hence lead to inaccurate locations of compatibility issues.

One promising solution is to leverage the DroidRA [88, 89, 148] and JuCify [142] tools to mitigate the impact of reflective calls and native code, respectively. Currently, *RepairDroid* is only aware of the most common expression types defined in Soot.

Manual testing: We have only evaluated the correctness of device-specific compatibility issue repairings through manual confirmation. This is because we cannot obtain all of the relevant devices for full testing. Such manual processes are, however, known to be error-prone. To mitigate the threat, we have cross-validated the results. We have also released our tool and dataset for public reference.

6.3 Future Work

As for future work, we have identified a few key research directions:

- For our first task of big code reliability analysis for malware detection approaches, we experimentally show that sample duplication indeed impacts the performance of machine learning-based Android malware detection approaches. For future work, we are interested in exploring more factors that may cause bias: (1) **a more realistic setting for preparing the training and testing datasets for evaluating the impact of sample duplication for supervised learning approaches could be explored**, such as taking app release time into consideration; (2) future work could focus on **improving the feature extraction process** by leveraging semantic features extracted based on Android apps' graph representations and advanced deep learning algorithms driven by neural networks, in order to better characterize the semantic features of Android apps.

- For our second task of API recommendations for Android app development, there are several areas for future work. Firstly, it would be valuable to **explore the impact of app obfuscation on the effectiveness of the APIMatchmaker approach** and to **introduce automated approaches to validate app descriptions** before applying them to the approach. Additionally, since Kotlin is becoming increasingly popular for developing Android apps, it would be beneficial to **extend the approach to support Kotlin-based app projects**. Furthermore, **conducting user studies** would help us better understand if the approach fulfills the actual demand of Android app developers.

  Moreover, GUI is a ubiquitous feature for all mobile apps, event-centric programs driven by rich graphical user interface interactions with users. One of the significant complexities of implementing mobile app GUIs is managing the complicated

and intertwined callback events from user interaction. Fortunately, functional APIs are capable of helping developers implement the functionalities in callback methods. Nevertheless, it is still a time-consuming task to identify and correctly use the appropriate functional APIs to fulfill the callback methods for implementing feature requirements [178]. To help developers efficiently and effectively implement these callback methods of GUI components, we have also presented another tool named **Icon2Code** [184, 185] to recommend API usages to assist Android app developers in implementing the callback functions of iconic GUI components. Icon2Code leverages icon image files and their alternative text to locate similar icons that are closest to the active icon under development. It then employs a collaborative filtering algorithm with an encoding matrix and rating algorithms to obtain an output of recommended APIs to call in the event handler code, as well as usage samples from existing apps. In addition to GUI development, there are more fine-grained development scenarios. For future work, we plan to **provide accurate and adaptive API recommendation approaches for each scenario**. Investigating and developing adaptive API recommendation methods that can be tailored to specific development scenarios would be a valuable direction for future research.

- In our third research on compatibility issue repair, one limitation of RepairDroid is its lack of awareness of multi-threading features, which can result in inaccurate analysis results. To address this limitation, we plan to **investigate the integration of more advanced analysis techniques**, such as those implemented in ThreadSafe, to handle multi-threading features for future work. Additionally, the manual confirmation process for evaluating the correctness of device-specific compatibility issue repairings can be error-prone. To improve the accuracy of the evaluation process, we intend to **explore automated evaluation techniques,** such as leveraging device emulators. Moreover, we aim to extend RepairDroid's ability to handle more of Soot's

expression types to **generate executable statements for unusual expressions**. As new types of compatibility issues emerge, we plan to **add more issue types to the app patch description language proposed with RepairDroid** to guide the tool in locating and repairing more types of compatibility issues effectively.

6.4 Summary

In summary, this Ph.D. thesis focuses on improving app quality with big code analysis-based analysis techniques and addresses three research objectives. The first study focused on big code reliability analysis and explored the impact of **sample duplication** on machine learning-based Android malware detection. The second study introduced **API-Matchmaker**, a tool for recommending API usages by learning directly from similar real-world Android apps. The third study presented **RepairDroid**, a tool for automatically repairing compatibility issues in published Android apps. Our research provides insights and tools for app markets, researchers, and developers to enhance the effectiveness and reliability of malware detection and high-quality Android app development. We plan to continue proposing novel solutions for improving the quality of Android apps in the future.

References

[1] Duckduckgo android app, 2017. https://github.com/duckduckgo/Android.

[2] Towards automatically repairing compatibility issues in published android apps, 2021. https://zenodo.org/record/5430715.

[3] 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] Charu C Aggarwal, Mansurul A Bhuiyan, and Mohammad Al Hasan. Frequent pattern mining algorithms: A survey. In *Frequent pattern mining*, pages 19–64. Springer, 2014.

[5] Mohammed S Alam and Son T Vuong. Random forest classification for detecting android malware. In *2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*, pages 663–669. IEEE, 2013.

[6] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153. ACM, 2019.

[7] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 21(1):183–211, Feb 2016. https://doi.org/10.1007/s10664-014-9352-6 doi:10.1007/s10664-014-9352-6.

[8] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Are your training datasets yet relevant? In Frank Piessens, Juan Caballero, and Nataliia Bielova, editors, *Engineering Secure Software and Systems*, pages 51–67, Cham, 2015. Springer International Publishing. URL: https://doi.org/10.1007/978-3-319-15618-7_5.

[9] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2901739.2903508, https://doi.org/10.1145/2901739.2903508 doi:10.1145/2901739.2903508.

[10] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[11] APIMatchmaker. *The dataset for APIMatchmaker*. 2021. URL: https://zenodo.org/record/5812605, https://doi.org/10.5281/zenodo.5812605 doi:10.5281/zenodo.5812605.

[12] Marco Aresu, Davide Ariu, Mansour Ahmadi, Davide Maiorca, and Giorgio Giacinto. Clustering android malware families by http traffic. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 128–135. IEEE, 2015.

[13] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA*, volume 14, pages 23–26, 2014.

[14] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 13–24. IEEE, 2017.

[15] 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.

[16] Zarni Aung and Win Zaw. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

[17] 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.

[18] Shams Azad, Peter C Rigby, and Latifa Guerrouj. Generating api call rules from version history and stack overflow posts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(4):1–22, 2017.

[19] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 9, pages 8–11, 2009.

[20] Daniel Bilar. Opcodes as predictor for malware. volume 1, pages 156–168. Citeseer, 2007.

[21] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[22] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 360–369. IEEE, 2016.

[23] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 114–126, 2009.

[24] Peter Brusilovski, Alfred Kobsa, and Wolfgang Nejdl. *The adaptive web: methods and strategies of web personalization*, volume 4321. Springer Science & Business Media, 2007.

[25] Evgeny Burnaev and Dmitry Smolyakov. One-class svm with privileged information and its application to malware detection. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 273–280. IEEE, 2016.

[26] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. A large-scale study of application incompatibilities in android. In *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*, 2019.

[27] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. Mobile malware detection using op-code frequency histograms. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 4, pages 27–38. IEEE, 2015.

[28] Joymallya Chakraborty, Tianpei Xia, Fahmid M Fahid, and Tim Menzies. Software engineering for fairness: A case study with hyperparameter optimization. *arXiv preprint arXiv:1905.05786*, 2019.

[29] Tanmoy Chakraborty, Fabio Pierazzi, and VS Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 17(2):262–277, 2017.

[30] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected api subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[31] Annie Chen. Context-aware collaborative filtering system: Predicting the user's preference in the ubiquitous computing environment. In *International Symposium on Location-and Context-Awareness*, pages 244–253. Springer, 2005.

[32] Chunyang Chen and Zhenchang Xing. Similartech: automatically recommend analogical libraries across different programming languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 834–839, 2016.

[33] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 112–124, 2020.

[34] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael O'Boyle. M3: Semantic api migrations. *arXiv preprint arXiv:2008.12118*, 2020.

[35] Wikipedia contributors. Sequential minimal optimization, July 2020. URL: `https://en.wikipedia.org/wiki/Sequential_minimal_optimization`.

[36] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Evolution and fragilities in scripted gui testing of android applications. In *Proceedings of the 3rd International Workshop on User Interface Test Automation. ACM*, 2017.

[37] David Curry. Android statistics (2023), 2023. URL: `https://www.businessofapps.com/data/android-statistics/`.

[38] 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.

[39] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, 2018.

[40] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *International*

Conference on Security and Privacy in Communication Systems, pages 172–192. Springer, 2018.

[41] 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*, 13(8):1890–1905, 2018.

[42] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. Graph embedding based familial analysis of android malware using unsupervised learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 771–782. IEEE, 2019.

[43] Ming Fan, Wenying Wei, Xiaofei Xie, Yang Liu, Xiaohong Guan, and Ting Liu. Can we trust your explanations? sanity checks for interpreters in android malware analysis. *arXiv preprint arXiv:2008.05895*, 2020.

[44] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 204–215, 2019.

[45] Ivan Firdausi, Charles Lim, Alva Erwin, and Anto Satriyo Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 second international conference on advances in computing, control, and telecommunication technologies*, pages 201–203. IEEE, 2010.

[46] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 254–265, 2016.

[47] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 2016.

[48] Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. Negative results on mining crypto-api usage rules in android apps. In *The 16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.

[49] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Should you consider adware as malware in your study? In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*, 2019.

[50] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[51] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability (TRel)*, 70(1):212–230, 2021.

[52] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):11, 2018.

[53] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Codekernel: A graph kernel based approach to the selection of api usage examples. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 590–601. IEEE, 2019.

[54] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[55] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. Androevolve: Automated android api update with data flow analysis and variable denormalization. *arXiv preprint arXiv:2011.05020*, 2020.

[56] Stefanus Agus Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automatic android deprecated-api usage update by learning from single updated example. *arXiv preprint arXiv:2005.13220*, 2020.

[57] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177. IEEE, 2018.

[58] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. Diversified third-party library prediction for mobile app development. *IEEE Transactions on Software Engineering*, 2020.

[59] Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, and Nikhil Ranade. Security toolbox for detecting novel and sophisticated android malware. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 733–736. IEEE Press, 2015.

[60] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo

ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260. IEEE, 2015.

[61] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 187–198, 2013.

[62] Yangyu Hu, Haoyu Wang, Yajin Zhou, Yao Guo, Li Li, Bingxuan Luo, and Fangren Xu. Dating with scambots: Understanding the ecosystem of fraudulent dating applications. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[63] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 532–542, 2018.

[64] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 293–304, 2018.

[65] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 425–435. IEEE, 2017.

[66] Paul Irolla and Alexandre Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, 14(3):245–249, 2018.

[67] John Jacobellis, Na Meng, and Miryung Kim. Lase: an example-based program transformation tool for locating and applying systematic edits. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1319–1322. IEEE, 2013.

[68] Quentin Jerome, Kevin Allix, Radu State, and Thomas Engel. Using opcode-sequences to detect malicious android applications. In *2014 IEEE International Conference on Communications (ICC)*, pages 914–919. IEEE, 2014.

[69] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*, 12(1):34–46, 2016.

[70] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Semantic patches for java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[71] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24:S48–S59, 2018.

[72] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy - a code-to-code search engine. In *The 40th International Conference on Software Engineering (ICSE 2018)*, 2018.

[73] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2018.

[74] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. iFixR: Bug report driven program repair. In *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325. ACM, 2019. https://doi.org/10.1145/3338906.3338935 doi:10.1145/3338906.3338935.

[75] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, 2020. https://doi.org/10.1007/s10664-019-09780-z doi:10.1007/s10664-019-09780-z.

[76] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[77] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering*, 2020.

[78] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference*, pages 601–614, 2018.

[79] Chenglin Li, Keith Mills, Di Niu, Rui Zhu, Hongwen Zhang, and Husam Kinawi. Android malware detection based on factorization machine. *IEEE Access*, 7:184008–184019, 2019.

[80] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, 2018.

[81] Li Li. Mining androzoo: A retrospect. In *The Doctoral Symposium of 33rd International Conference on Software Maintenance and Evolution (ICSME-DS 2017)*, 2017.

[82] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*, 2015.

[83] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[84] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, 2018.

[85] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering (TSE)*, 2019.

[86] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter values of android apis: A preliminary study on 100,000 apps. In *2016 IEEE 23rd Inter-*

national Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 584–588. IEEE, 2016.

[87] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[88] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[89] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *The 31st IEEE/ACM International Conference on Automated Software Engineering, Demo Track (ASE 2016)*, 2016.

[90] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[91] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[92] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264, 2018.

[93] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Cda: Characterising deprecated android apis. *Empirical Software Engineering*, pages 1–41, 2020.

[94] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.

[95] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology*, 32(6):1108–1124, 2017.

[96] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

[97] 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.

[98] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2021. https://doi.org/10.1109/TSE.2018.2884955 doi:10.1109/TSE.2018.2884955.

[99] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault

localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*, pages 102–113. IEEE, 2019. https://doi.org/10.1109/ICST.2019.00020 doi:10.1109/ICST.2019.00020.

[100] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 456–467. IEEE, 2019. https://doi.org/10.1109/SANER.2019.8667970 doi:10.1109/SANER.2019.8667970.

[101] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42. ACM, 2019. https://doi.org/10.1145/3293882.3330577 doi:10.1145/3293882.3330577.

[102] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. LSRepair: live search of fix ingredients for automated program repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, pages 658–662. IEEE, 2018. https://doi.org/10.1109/APSEC.2018.00085 doi:10.1109/APSEC.2018.00085.

[103] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021. https://doi.org/10.1016/j.jss.2020.110817 doi:10.1016/j.jss.2020.110817.

[104] Kui Liu, Shangwen Wang, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 615–627. ACM, 2020. https://doi.org/10.1145/3377811.3380338 doi:10.1145/3377811.3380338.

[105] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. Identifying and characterizing silently-evolved methods in the android api. In *The 43rd ACM/IEEE International Conference on Software Engineering, SEIP Track (ICSE-SEIP 2021)*, 2021.

[106] Pei Liu, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. Androzooopen: Collecting large-scale open source android apps for the research community. In *The 2020 International Conference on Mining Software Repositories, Data Track (MSR 2020)*, 2020.

[107] Tianming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. Dapanda: Detecting aggressive push notification in android apps. In *The 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019.

[108] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé F Bissyandé, and Jacques Klein. Maddroid: Characterising and detecting devious ad content for android apps. In *The Web Conference 2020 (WWW 2020)*, 2020.

[109] 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 inter-*

national conference on software engineering companion, pages 653–656, 2016.

[110] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Network and Distributed Systems Security Symposiym (NDSS)*, San Diego, CA, 2017.

[111] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308. ACM, 2017.

[112] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[113] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):1–30, 2013.

[114] Na Meng, Miryung Kim, and Kathryn S McKinley. Lase: locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 502–511. IEEE, 2013.

[115] Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, Paul Miller, and Ziming Zhao. Dandroid: A multi-view discriminative adversarial network for obfuscated android malware detection. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 353–364, 2020.

[116] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 880–890. IEEE, 2015.

[117] Annamalai Narayanan, Guozhu Meng, Liu Yang, Jinliang Liu, and Lihui Chen. Contextual weisfeiler-lehman graph kernel for malware detection. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4701–4708. IEEE, 2016.

[118] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, 2016.

[119] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. Focus: A recommender system for mining api function calls and usage patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1050–1060. IEEE, 2019.

[120] Phuong T Nguyen, Juri Di Rocco, Claudio Di Sipio, Davide Di Ruscio, and Massimiliano Di Penta. Recommending api function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*, 2021.

[121] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Recommending api usages for mobile apps with hidden markov model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 795–800. IEEE, 2015.

[122] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Learning api usages from bytecode: a statistical approach. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 416–427. IEEE, 2016.

[123] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, 2016.

[124] Haoran Niu, Iman Keivanloo, and Ying Zou. Api usage pattern recommendation for software development. *Journal of Systems and Software*, 129:127–139, 2017.

[125] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[126] Cyrus Omar, Young Seok Yoon, Thomas D LaToza, and Brad A Myers. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 859–869. IEEE, 2012.

[127] Takayuki Omori, Hiroaki Kuwabara, and Katsuhisa Maruyama. Improving code completion based on repetitive code completion operations. *Information and Media Technologies*, 10(2):210–225, 2015.

[128] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In *NDSS*, 2017.

[129] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*, pages 300–305. IEEE, 2013.

[130] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 729–746, Santa Clara, CA, August 2019. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury`.

[131] Roberto Perdisci and ManChon U. Vamo: towards a fully automated malware clustering validity analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 329–338, 2012.

[132] Yihao Qin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. On the impact of flaky tests in automated program repair. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 295–306. IEEE, 2021.

[133] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 357–367. IEEE, 2016.

[134] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 349–359. IEEE, 2016.

[135] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[136] Douglas A Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741:659–663, 2009.

[137] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[138] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[139] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415. IEEE, 2017.

[140] Pascal Roos. Fast and precise statistical code completion. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 757–759. IEEE, 2015.

[141] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[142] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. *arXiv preprint arXiv:2112.10469*, 2021.

[143] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, pages 289–298. Springer, 2013.

[144] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

[145] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, 2019.

[146] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Effective smart completion for javascript. *Technical Report RC25359*, 2013.

[147] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[148] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. Taming reflection: An essential step towards whole-program analysis of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

[149] Zhongbin Sun, Jingqi Zhang, Heli Sun, and Xiaoyan Zhu. Collaborative filtering based recommendation of sampling methods for software defect prediction. *Applied Soft Computing*, 90:106163, 2020.

[150] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 46(11):1200–1219, 2018.

[151] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.

[152] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. Automated deprecated-api usage update for android apps: How far are we? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 602–611. IEEE, 2020.

[153] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191. IEEE, 2013.

[154] Ferdian Thung, Richard J Oentaryo, David Lo, and Yuan Tian. Webapirec: Recommending web apis to software projects via personalized ranking. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):145–156, 2017.

[155] Ferdian Thung, Shaowei Wang, David Lo, and Julia Lawall. Automatic recommendation of api methods from feature requests. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 290–300. IEEE, 2013.

[156] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting patch correctness based on the similarity of failing test cases. *ACM Transactions on Software Engineering and Methodology*, 2022.

[157] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 981–992. IEEE, 2020. https://doi.org/10.1145/3324884.3416532 doi:10.1145/3324884.3416532.

[158] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[159] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *International Conference on Rewriting Techniques and Applications*, pages 357–361. Springer, 2001.

[160] Haoyu Wang, Hongxuan Liu, Xusheng Xiao, Guozhu Meng, and Yao Guo. Characterizing android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 280–292. IEEE, 2019.

[161] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. Rmvdroid: towards a reliable android malware dataset with app metadata. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 404–408. IEEE Press, 2019.

[162] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. Exploring how deprecated python library apis are (not) handled. In *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[163] 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'17), pages 252–276, Bonn, Germany, 2017. Springer.

[164] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.

[165] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 878–888. IEEE, 2019.

[166] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. Clear: contrastive learning for api recommendation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 376–387, 2022.

[167] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 194–203. IEEE, 2004.

[168] Junwei Wu, Liwei Shen, Wunan Guo, and Wenyun Zhao. Code recommendation for android development: how does it work and what can be improved? *Science China Information Sciences*, 60(9):092111, 2017.

[169] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and Software Technology*, 75:17–25, 2016.

[170] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. How android devel-

opers handle evolution-induced api compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 886–898. IEEE, 2020.

[171] Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, and Xudong Liu. A hybrid approach to code reviewer recommendation with collaborative filtering. In *2017 6th International Workshop on Software Mining (SoftwareMining)*, pages 24–31. IEEE, 2017.

[172] Zhiwu Xu, Kerong Ren, Shengchao Qin, and Florin Craciun. Cdgdroid: Android malware detection based on deep learning using cfg and dfg. In *International Conference on Formal Engineering Methods*, pages 177–193. Springer, 2018.

[173] Guowei Yang, Jeffrey Jones, Austin Moninger, and Meiru Che. How do android operating system updates impact apps? In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 156–160. IEEE, 2018.

[174] Wei Yang, Mukul R Prasad, and Tao Xie. Enmobile: Entity-based characterization and analysis of mobile malware. In *Proceedings of the 40th International Conference on Software Engineering*, pages 384–394, 2018.

[175] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. Characterizing malicious android apps by mining topic-specific data flow signatures. *Information and Software Technology*, 90:27–39, 2017.

[176] Huan Yu, Xin Xia, Xiaoqiong Zhao, and Weiwei Qiu. Combining collaborative filtering and topic modeling for more accurate android mobile app library recommendation. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, pages 1–6, 2017.

[177] Jian Yu, Miin-Shen Yang, and E Stanley Lee. Sample-weighted clustering methods. *Computers & mathematics with applications*, 62(5):2200–2208, 2011.

[178] Weizhao Yuan, Hoang H Nguyen, Lingxiao Jiang, and Yuting Chen. Libraryguru: Api recommendation for android developers. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 364–365, 2018.

[179] Weizhao Yuan, Hoang H Nguyen, Lingxiao Jiang, Yuting Chen, Jianjun Zhao, and Haibo Yu. Api recommendation for event-driven android application development. *Information and Software Technology*, 107:30–47, 2019.

[180] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[181] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 956–961, 2016.

[182] Jingtang Zhang, Kui Liu, Dongsun Kim, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. Revisiting test cases to boost generate-and-validate program repair. In *Proceedings of the 37th International Conference on Software Maintenance and Evolution*, pages 35–46. IEEE, 2021.

[183] Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. Toolcoder: Teach code generation models to use apis with search tools. *arXiv preprint arXiv:2305.04032*, 2023.

[184] Yanjie Zhao, Li Li, Xiaoyu Sun, Pei Liu, and John Grundy. Icon2code: Recommending code implementations for android gui components. *Information and Software Technology*, 138:106619, 2021.

[185] Yanjie Zhao, Li Li, Xiaoyu Sun, Pei Liu, and John Grundy. Code implementation recommendation for android gui components. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 31–35, 2022.

[186] XU Zhiwu, Kerong Ren, and Fu Song. Android malware family classification and characterization using cfg and dfg. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 49–56. IEEE, 2019.

[187] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.

[188] Jing Zhou and Robert J Walker. Api deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 266–277, 2016.

[189] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.