An automatic authorship attribution technique for Android applications

by

Hugo F. Gonzalez Robledo

M.Sc. in Computer Science, Technological Institute of San Luis Potosi, Mexico, 2005.
Bachelor of Software Engineering, Technological Institute of San Luis Potosi, Mexico, 2000.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

In the Graduate Academic Unit of Faculty of Computer Science

Supervisor(s):	Ali A. Ghorbani, Ph.D., Faculty of Computer Science, UNB
	Natalia Stakhanova, Ph.D., Faculty of Computer Science, UNB
Examining Board:	Rodney Cooper, Professor, Faculty of Computer Science, UNB, Chair
	Ronxing Lu, Ph.D., Faculty of Computer Science, UNB
	Brent Petersen, Ph.D., Department of Electrical and Computer Engineering, UNB
External Examiner:	Nur Zincir-Heywood, Ph.D., Faculty of Computer Science, Dalhousie University

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2017

©Hugo F. Gonzalez Robledo, 2017

Abstract

In recent years, mobile devices have overcome desktop computers in usage. The most popular system for mobile devices is Android; hence, it is the most targeted by malicious applications. Typical signature detection systems had been proved to be ineffective for targeted attacks; therefore, other approaches should be explored. It is a common believe that only a handful of skilled authors exists to produce new malware, the overwhelming amount of current samples are adaptations, evolution, and modifications of an original strain. It is possible to associate a piece of malware with the adversary behind it by employing authorship attribution techniques. In other words, we believe that personal preferences of mobile software developers (including malware developers) leave traces in the produced software, i.e., apps. These preferences are unique to each developer. Discovering these preferences would allow us to attribute an unknown app to an author. In this thesis, we proposed, developed, and evaluated a framework to automatically discover and create new author profiles of unknown Android apps on an open world problem. To accomplish this goal, we prepare and release two new Android datasets, one

for legitimate and the other for malicious authors. Also, we define a set of stylistic features extracted from Android apk files to be used by the framework. The heart of the framework is an algorithm to discover and create new author profiles from a stream of apps; it uses machine learning boosting techniques to calculate prediction probability of new samples, then group them and measure their cosine similarity, and finally discard outliers.

We employ a 5-fold cross-validation over a fixed set of apps with over a 99% of accuracy, precision, and recall. For unknown apps, we measure how many were attributed and how many were attributed to suspicious profiles. In our extensive experiments we discover five apps in the official market related to ransomware profile that were not detected by VirusTotal service as malicious immediately, but a couple of weeks later. We also present a use case where our solution is employed to evaluate the threat landscape in the official Android market.

Dedication

To God, the Almighty Creator.

To Nadia, the love of my life, for all the support, love, care and sharing during this journey.

To Jimena and Daniela, two little big reasons to keep going, for all the love, lessons and those enjoyable moments.

To my parents Rosario and Jorge, for all the love, care and encouraging during my hole life.

To my ancestors, because without them, I won't be here being who I am.

Acknowledgements

Foremost, I would like to thank my PhD advisers, Dr. Ali A. Ghorbani and Dr. Natalia Stakhanova, for the continuous support and guidance during my study and research these past four years. Thank you for your patience, motivation, enthusiasm and support; even when you were very busy with your appointments, you always found time to answer e-mails, have quick walking talks or discuss about new ideas and projects.

I also would like to thank the rest of my thesis committee: Professor Cooper, Professor DeDourek, Dr. Lu, Dr. Petersen and Dr. Zincir-Heywood for their time, encouragement, insightful comments, and interesting questions.

I thank my research colleagues at former Information Security Centre of Excellence (ISCX) (now Canadian Institute for Cybersecurity (CIC)) for the stimulating discussions and exciting projects we worked on together. Especially to Abdulla, Amirhosseing, Andi Fitriah for Android related discussions, ideas, projects and experiments. Thanks to Ratinder, Alina and Yan for the discussions, ideas and hard work in other several projects. Thanks to all the students and staff that converged in the "lab", with more or less interaction,

I'm sure that I learned something from you, and I hope you learned something from me. Also my gratitude to the fellas that helped me to rediscover my passion for the volleyball, organizing the team to play regularly at UNB and outside of the University.

I should express my gratitude to my previous mentors, friends, colleagues and family members who encourage me to continue my studies abroad. Thanks to "The Honeynet Project" and its members, who opened my view and helped me to view the world from a different perspective.

I need to thank my family for all the support during these years: my parents, my wife and my kids.

Last but not the least, I would like to thank Mexico's Government for my PROMEP scholarship during the first three years of my PhD studies. And of course all the support from Universidad Politecnica de San Luis Potosi.

Table of Contents

Abstract ii

Dedication iv

Acknowledgments v

Table of Contents vii

List of Tables ix

List of Figures xi

1 Introduction 1

1.1 Problem Statement 5

1.2 Research Motivations 6

1.2.1 Challenges 8

1.3 Contributions 10

1.4 Thesis outline 13

2 Background and related work 15

2.1 Authorship attribution process 16
2.1.1 Features 17
2.1.2 Representations 20
2.1.3 Authorship attribution methods 25
2.1.3.1 Profile-based approach 27
2.1.3.2 Instance-based approach 28
2.1.3.3 Hybrid approach 29
2.2 Basic introduction to Android platform 30
2.3 Related work 36
2.3.1 Source code authorship attribution 37
2.3.2 Binary code authorship attribution 41
2.3.3 Malware analysis 43
2.3.4 Android malware analysis 45
2.4 Concluding remarks 48

3 An automatic authorship attribution technique for Android applications 51

3.1 Filter 53
3.2 Feature extraction 55
3.2.1 Features extracted. 58
3.3 Feature representation. 69
3.4 Attribution and Discovery approach 73
3.5 Prototype implementation 79

3.5.1 Architectural view 79
3.5.2 Implementation challenges 87
3.5.3 Technical details of developed software components . . 89
3.6 Summary 93

4 Experimental results **95**
4.1 Datasets 96
4.2 Analysis of the authors' datasets 102
4.3 Evaluation metrics 106
4.4 Reference profiles validation. 108
4.5 Tuning thresholds 114
4.6 Evaluation of emergent-profiles. 121
4.7 Experimenting with attribution only approach. 129
4.8 Concluding remarks 137

5 Conclusions and future work **139**

Bibliography **144**

A Evolution of studies about Authorship Attribution **167**
A.1 Bibliography 167

Vita

List of Tables

2.1 Summary of features and some examples. 21

3.1 Android packers detected. 55
3.2 Groups of features. 56
3.3 Directly extracted features. 58
3.4 Inferred features from string section. 59
3.5 Features composed by opcodes. 62
3.6 Features composed from a .dex file. 65
3.7 Features non related to a .dex file. 68
3.8 Features type and representations. 70

4.1 Datasets. 97
4.2 Relation between experiment stages and datasets. 102
4.3 Experimental results from filter step for legitimate and malicious authors. 104
4.4 Filters created with CommonCode tool. 105
4.5 Experimental results from created filters over legitimate and malicious authors. 105

4.6 Experimental results of signature distribution among legitimate and malicious authours. 106
4.7 Validation on legitimate authors dataset. 111
4.8 Validation on malicious authors dataset. 112
4.9 Validation over all authors. 113
4.10 Experimental results from discovery approach considering malware families as coming from same author. 123
4.11 Experimental results for emergent-profiles considering malware families as coming from same author. (First Part). 125
4.12 Experimental results for emergent-profiles considering malware families as coming from same author. (Second Part). . . 126
4.13 Experimental results from discovery approach over legitimate apps. 129
4.14 Experimental results for authorship attribution over a large unknown suspicious stream of apps. 131
4.15 Experimental results for authorship attribution over a large unknown stream of apps. 132
4.16 Experimental results from attribution over new families' apps. 135

List of Figures

2.1 General steps to the process of software authorship attribution. 16

2.2 Android package (.apk) 31

2.3 Android application project build process 33

2.4 The structure of a typical plain .dex file. 35

2.5 Evolution of studies on software authorship attribution. 38

3.1 The flow of proposed approach to work on Android apps authorship attribution. 52

3.2 Overview of filter phase. 53

3.3 The process of building Bloom filters 61

3.4 StructSignature technique description. 65

3.5 Creation of feature vector. 72

3.6 Attribution and Discovery approach. 74

3.7 Five categories holding different modules of our system. 81

3.8 External modules related each package of our system. 81

3.9 Client-Server architectural view of the proposed framework. . 83

3.10 Overview of the attribution process. 84

3.11 Overview of the feature union process. 85

3.12 Python code for producing a feature vector from an Android app. 86
3.13 Visualization of a .dex file structure. 90
3.14 Modified example of DexStrings output. 91
3.15 Overview of the path to parse class information to binary code. 91

4.1 Legitimate authors dataset grouped by Droidkin. 100
4.2 Experimental results with various benchmarkThresholds . . . 115
4.3 Graphical representation of defined bucket edges. 117
4.4 Experimental results with defined bucket edges. 118
4.5 Experimental results with various incrementalThresolds. . . . 119
4.6 Experimental results with different outliers approaches. 120
4.7 Experimental results for emergent-profile evaluation with different configurations. 122
4.8 Malware analysis results from more than three years ago in VirusTotal service. 130
4.9 Droidkin results for the two larger groups of apps attributed to DroidKungfu malware related profiles. 133
4.10 A preview of the use case system. 136

Chapter 1

Introduction

"It is not a malware problem, it is an author problem."

Anonymous

The number of actual threats in cyber space related to malware keeps growing: antivirus companies receive thousands of malware variants every day. According to the 2013 report "Global Corporate IT Security Risks", Kaspersky Labs were receiving up to 200,000 unique samples daily [89]. In 2014, mobile malware continued its unprecedented growth increasing by 600%. Gdata [33] reported half of mobile malware been financially motivated through different mechanisms. In the last quarter of 2015, 72% of unique mobile malware was seen by McAfee labs [117], leading to almost two and a half million unique samples. The company also reported a dramatical increase in sophistication and complexity of mobile malware.

The traditional approaches predominantly based on recognizing well-documented

malware samples are struggling to cope with this unprecedented rate of growth in the number of malware mainly owing to several concerns. First of all, traditional detection methods are mostly based on manually crafted signatures, describing known malware samples. The second problem that traditional defenses can not overcome is the widespread use of obfuscation techniques that allow a malicious coder to modify the appearance of a malware sample while leaving its functionality intact. Much of the software employed to produce obfuscation includes some sort of protection to prevent a binary file being analyzed, or just make it harder to understand when performing an analysis. Thirdly, a rise of code obtained dynamically at run time, or through updates for apps, is used to evade detection, creating malicious capabilities from benign software at some stage of its lifetime.

Finally, malware authors employ a variety of different techniques to probe an analysis environment. These techniques hide malware functionality, producing misleading behavior or simply doing nothing in a suspected analysis environment. As an example modern malware is able to detect the presence of debuggers, virtual machines, and special configurations, and later decide how to behave.

As a result malware strains go undetected. The inadequacy of traditional approaches exposes an urgent need for alternative defenses.

In this thesis, we propose to explore an alternative route and focus our attention on the source of the problem, i.e., malware authors. Our main hypothesis is defined as follows: *personal preferences of mobile software developers (in-*

cluding malware developers) leave traces in the produced software, i.e.,apps. These preferences are unique to each developer. Discovering these preferences would allow us to recognize a developer behind an app, or in other words, attribute an unknown app to an author.

Authorship attribution approaches aim to identify the original author of a text, i.e., extract the unique characteristics of an author's writing/coding style, based on a set of stylistic discriminators and previous known work of that author.

In a traditional setting author attribution relies heavily on the linguistic information found in the document. The main premise of attribution lies in the fact that every author has a unique style that reflects the author's preferences, habits, and knowledge. This style acts as a fingerprint giving hints on the author's identity. In the linguistic domain, such a fingerprint includes the use of punctuation, expressions, common and rare words, and even the semantics. In the context of software, we can draw an analogy with the author's preferences with regard to development environments, data structures, system calls, programming language, libraries, and even function names and parameter values. Authorship attribution analysis is often seen as a practical technique in clone detection, similarity matching, software evolution, plagiarism, digital forensics and copyright infringements.

When analyzing malware, authorship attribution provides more insights about the current threats, and can offer clues about actors behind the code. Knowing how big or organized are the groups performing cybercrimes; whether it

was a criminal act, an act of war or just a cyberbased vandalism; and if we can expect further attacks or it it was a one-off, help us to describe the actual level of threat in the landscape. Last but not least, malware attribution leads to an increased risk for adversaries. Being in the open not only reduces profit, but also attracts more attention. As in the case of the Hacking Team organization, this attention got them hacked, completely exposed and out of business for a while. Hacking Team Services offered surveillance software and infrastructure to governments and law enforcers. Their software had been considered as malware because of its nature and behavior; as an example, their catalog includes mobile malware that can be installed without user knowledge, it tracks users and exfiltrates information from mobile devices to remote servers.

Malware authorship attribution also leads to improved defenses with an inverse relationship between commoditization and detection. Generic malicious code, also known as commodity malware, is difficult to attribute, but should be easier to detect using traditional methods; i.e., a routine to retrieve information from a web server that is widely known and available to implement. Specialized, carefully crafted malicious code is difficult to detect using traditional methods; however, it opens an opportunity for authorship attribution methods.

1.1 Problem Statement

Today the amount of malware produced and released is overwhelming the capacities of infrastructure for both practitioners and researchers. Besides the efforts to automatically conduct an analysis, the time and resources needed are growing. Also the number of legitimate mobile apps produced and released is growing, i.e., number of new mobile apps listed each day in the official Android market approaches hundreds.

Recognizing a new serious threat in this high volume of malware samples is a complicated and time consuming process. Most of the time a new threat is detected when it already has caused damage. As an example, a malware campaign named "Gooligan", had distributed malicious apps on the official Android market and compromised more than 1 million Google accounts before being detected [24].

The current detection model that focuses on the analysis and characterizations of malware samples is useful for commodity malware; however, it is becoming useless for sophisticated or specialized malware. In the current model, researchers need a sample of a new malware to analyze it, then they create signatures for it. New malware specimens can be under the radar for long time. Now we should improve the actual detection model, focusing on characterizing the origin of the malware through authors' profiles. With this approach, instead of needing a previous sample of malware, we need an author profile to attribute new malware specimens.

1.2 Research Motivations

Despite the interest of the research community in author attribution in the software domain, there is still a gap in research focused on analysis of software binary code. This is present especially in the malware domain where source code is not always available. Also, current methods conduct analysis from malware samples' perspective, not from malware authors' perspective.
Some of the benefits of analyzing malware authors through authorship attribution can be summarized as follows:

- Authorship identification. With the authorship attribution we can tie an author or a group to a specific piece of malware. We can follow the development of malware by a person of interest.

- Authorship clustering. We can group similarly written malware, this does not mean specifically by family, functions or goals, but by author profile.

- Authorship evolution. We can track the evolution of a piece of malware, and potentially the evolution of the author's skills.

- Authorship profiling. We can generate technical profiles for authors or groups, and have a better understanding of what we could expect and predict for the immediate future when a new specimen is discovered in the wild.

With this approach we believe that we are able to help to attribute actual and future threats. To address future threats, this approach should be able to learn incrementally to discover new profiles in a open world problem scenario. Authorship attribution had been typically conceived as a closed world problem, in which we have a specific set of authors and attribution should be done to one of these authors. On the other hand, in a open world problem we are not restricted to a specific set of authors, nor to static profiles. Author profiles should be created and evolve over time to keep up with malware proliferation.

In recent years, mobile devices have overcome desktop computers in usage and media consumption. These days people trust their mobile phone more than their personal computer. People play games, communicate with others, buy articles, pay for services, perform bank operations and other tasks on a daily basis with their mobile devices. Therefore, personal and identifiable data are input and stored on the devices. These maybe the factors that have precipitated an unprecedented number of mobile malware attacks. These factors also lead users to consider mobile devices risky and under a high level of threat.

Mobile malware is a serious and emerging threat. Android is an open platform, very popular among developers and the most common for mobile devices; therefore, the majority of mobile malware affects this platform. The amount of Android malware is increasing at an alarming speed and malicious capabilities are constantly evolving. This evolution has been studied

and described with regard to the apps' objectives and behavior, but not from the authors' perspective. Several research works are being done on detecting and classifying Android malware using different perspectives and features; however, the goals of these works remains focused on the malware samples.

1.2.1 Challenges

Authorship attribution of malicious Android developers faces several challenges:

- Closed world problem. Authorship attribution is performed typically over a set of known authors. If we have an unknown work from an unknown author it is not possible to attribute with this approach.

- Lack of source code. Especially in the malware domain, there is only a limited amount of malware source code available. Malware is distributed typically in binary format; although decompilation is possible sometimes, it inserts new artifacts, and is not possible to recover useful information removed by compiler. Using an intermediate representation language will also insert artifacts and noise; it will not recover other information like variable names, resulting in a distorted version of the original source code.

- Code and library reusing. In malware and legitimate apps, library usage and code reuse are very common. Because of the Android platform design, a big piece of the binary file is composed of basic libraries. To

produce malware at a fast pace, authors usually improve or modify existing code, reusing objects and components with some tweaks to remain undetected. If a legitimate component is used regularly from malware, legitimate apps will rise flags if they also use the same component.

- Code obfuscation. Obfuscation techniques are often used to protect intellectual property or to make reverse engineering difficult; to accomplish their objectives code transformations and other changes are performed. These transformations might affect analysis results.

- Tool dependency. Feature extraction is a crucial step for authorship attribution. Some tools are used to extract features from binary software; results might be affected by the quality, consistency and resilience of the tools employed.

- Lack of datasets. Research in software authorship attribution suffers from the lack of open benchmark datasets. Researchers have to manually collect and label software from specific authors to conduct their experiments.

- Authors' evolving programming style. Authors' programming style keeps on changing and evolving over time. This evolution impacts created profiles since these profiles usually are considered static once created.

- Source of legitimate apps and legitimate authors. The official market has played a crucial role in the Android security ecosystem, but even with their security measures in place, Android malware has been found in the official market during recent years, so we can not consider all apps from this market as non-malicious apps.

1.3 Contributions

In this thesis we propose a new approach to tackle the open world problem of authorship attribution, where we are able to produce new profiles and attribute previously unseen authors. This approach includes a modular and extensible Android authorship attribution framework powered by a novel algorithm to incrementally discover and create authors' profiles from a stream of Android apps. From those apps we extract and represent tailored features for the goal of authorship attribution. This framework leads to improvements in the understanding of mobile malware threats through the analysis of current and historic Android malware samples.

Within this framework we provide the following contributions:

1. The creation of a **list of crafted features** that can be extracted from the binary code of Android apps and are the most suitable for authorship attribution. (See Table 3.8)

2. The development of **software** to extract and represent **proposed features** from an Android app to be employed in the proposed framework

(See Section 3.5.3). Evaluation of these features in the Android authorship attribution domain is also conducted (See Section 4.4).

3. The design and evaluation of an **algorithm** based on machine learning techniques to incrementally discover and learn new author's profiles, offering a solution for an open-world problem in authorship attribution (See Section 3.4).

4. The design of an **architecture** for the proposed framework inspired by an authorship attribution model (See Chapter 3 and Section 3.5.1). This framework incorporates features, algorithms and software from previous contributions. The framework should be modular and extensible to permit exploration of different combinations of features, representations and authorship attribution methods (See Section 3.5.1). It should employ advanced technology that permits scalability, security and usability while analyzing streams of apps.

5. The demonstration of the practical applicability of the proposed technique by: (a) developing a prototype of the proposed framework (Code available at [54]) (b) performing experiments to demonstrate the effectiveness of the proposed framework concerning identifying malware authors. (See Sections 4.4 and 4.5) (c) performing large scale experiments to evaluate the system's performance (See Section 4.7).

6. The creation of a dataset of Android malware authors and a dataset of legitimate Android authors. These have been made publicly avail-

able to the academic community during the course of this work. (See Section 4.1). Also, a new label classification based on author profiles has been released for a well known Android malware dataset. (See Section 4.7).

The following datasets were created and offered to the academic community:

1. Dataset comprising 1,734 legitimate Android apps from 43 authors. These apps were collected from eight different markets. This dataset contains information about 43 known authors with more than 20 apps in the markets, based on the assumption that similar serial numbers from certificates lead to same author (without considering public or leaked certificates).

2. Dataset containing 222 malware Android apps from 10 authors. These apps were collected from the Koodous [83] system, with at least 10 unique and malicious apps from each of the 10 different authors. These malware apps were identified in the Koodous system as installed in mobile devices. We use the same assumption that similar certificates lead to same author. As an addition for this dataset, we collected 136 apps from the Hacking Team group.

1.4 Thesis outline

The authorship attribution problem is briefly described in this Chapter, along with the actual increasing distribution and complexity of malware, especially in the Android platform. We presented our research motivations and challenges for this problem. Finally, we introduced our contributions.

In Chapter 2 we describe the process to perform authorship attribution from a general perspective. We explain types of features and their representations to be employed with different authorship attribution methods. We also present a basic introduction to the Android platform and the development process, which helps the reader to better understand chosen features for our proposed solution. Later in the same chapter we review related work in four domains that helped us to develop our approach. While hundreds of studies in these domains are available, we pay special attention to those studies which influence directly our research. In Section 2.4 we present three identified gaps in the literature that our research is filling: Closed world problem, lack of source code, and lack of datasets.

In Chapter 3 we explain our proposed solution, with special attention to features chosen for the authorship attribution process. We also explain the use of machine learning techniques that help us to accomplish our objective. To offer a solution for an open world problem, here we introduce a novel algorithm to discover and incorporate new author's profiles to the running model. We discuss our prototype implementation, architectural design and

some challenges.

Datasets, evaluation metrics, and experimental results are presented in Chapter 4. Section 4.1 offers details about datasets employed, highlighting authorship attribution datasets created for this study. Lack of ground truth with some datasets led us to develop three evaluation metrics for some experiments related to the creation and discovery of new author profiles; these metrics are explained in this chapter. Several experiments were conducted; descriptions and results are presented here. In closed world problem we obtain above 99% of accuracy, precision and recall when identifying authors from our created datasets. In open world problem experiments, with our worst case we attributed over 70% of apps, discovering and creating new author profiles. Also we discover five malicious apps in the official Android market, these apps were not detected by antivirus at that time but a couple of weeks later. Finally we present a use case where we are evaluating the threat landscape in the official Android market with our proposed framework.

We conclude our work with promising results, and offer ideas about future work in Chapter 5.

Chapter 2

Background and related work

"If I have seen further, it is by standing on the shoulders of giants."

Isaac Newton

In this thesis we propose an approach to perform authorship attribution on a set of Android apps. This approach finds a probable candidate author profile for a specific app. This information helps to asses the threat level of a suspicious app, using a completely different approach than antivirus products.

In this chapter we are covering a fundamental introduction to our two main topics, *authorship attribution process* and *Andriod apps*. We also offer a literature review related to this research. In Section 2.1 we are detailing the authorship attribution process and characteristics, explaining about features and methods in general. Section 2.2 introduces concepts about Android apps, Android binary format and the development process. Section 2.3 cov-

ers recent studies in software authorship attribution, binary authorship attribution, Android malware analysis and touches briefly on general malware analysis.

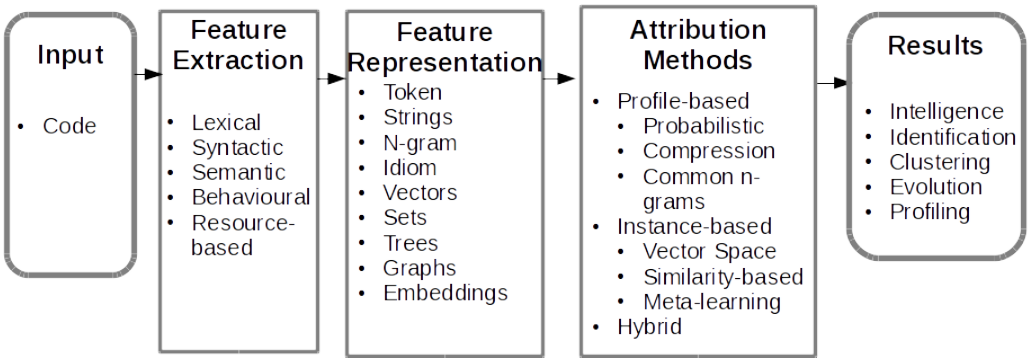

Figure 2.1: General steps to the process of software authorship attribution.

2.1 Authorship attribution process

The authorship attribution process aims to identify the original author of a piece or work, namely text or software. The traditional software authorship attribution process model follows three steps as shown in Figure 2.1:

1. Feature Extraction: In this step features are extracted from the author's work at binary or source code level.

2. Feature Representation: In this step the extracted features are represented in a suitable form. Sometimes additional intermediate steps are involved in preprocessing the data.

3. Authorship attribution: In this step a new text or software is assigned to an author. Different techniques to perform authorship attribution are available to complete this task.

A mix of available options in these steps will give us different results based on our goals, i.e., author identification, author clustering, author evolution or author profiling.
More details on features, feature representations and attribution methods are in the following subsections.

2.1.1 Features

In the history of authorship attribution research, employed features have always played a critical role. These features have varied from simple counting metrics in the "Software Science" [18, 65] to semantic oriented metrics such as control flow graphs [134] and software birthmarks [25, 23]. We could classify such features based on the level of extraction as follows: lexical features, syntactical features, semantic features and specific related features. Following their descriptions, Table 2.1 shows more examples of this classification.

Lexical features. These features are extracted from source or binary code when this code is treated as a simple sequence of tokens or characters. For example, the bag-of-words approach is a common technique that uses this type of feature. In these scenarios a token's function is not taken into account. For example, we count the length of the lines, the number of operands or the

number of spaces or vocabulary richness. Vocabulary richness attempts to measure the diversity of the vocabulary employed. Frequencies of the words or tokens is another example of these features. Note that lexical features do not take into account the semantics of the tokens. The goal here is to exploit the idea that individual authors tend to use the same or similar words to name functions, variables, or other resources.

Syntactic features. These features characterize code's external structural organization. At this level the focus is on the function of the token. For example, we could count the number of functions independently of the names, or the number of specific delimiters for a programming language. Instead of counting tokens, we are counting functions of the tokens, i.e., *add* token means *binary operation* in assembler programming language. The features used at this level from source code are fully related to the programming language, for example we could not count classes and methods on a program written in assembler programming language. Idioms and n-gram are examples of feature representation to use syntactic features. A different approach at syntactic level is to measure the warnings and errors when compiling source code. The goal with these features is to exploit the idea that individual authors tend to use similar syntactic patterns unconsciously.

Semantic features. These features express the logical flow of the code and give meaningful insight into the inner workings of the code. For example, a programming loop could be expressed in different syntactical ways, such as

for-loop, do-until, repeat-until, for-each; however, at a semantic level those forms represent a loop. Control flow graphs and functionality analysis are two methods to explore semantic features. The goal with these features is to exploit the idea that individual authors repetedly use the same logic when creating their programs.

Specific related features. These features are specific to type of files under analysis. Examples of these features are shown below:

- Behavioral features. In binary software authorship attribution the source code of a program is transformed into executable code that will perform some actions, in contrast to the literacy authorship attribution analysis where text are static. While the code is executed, there exists the possibility for new code to be obtained on the fly i.e., code obtained from external resources, functions being deobfuscated, or the usage of data structures as code. This fact implies that the static analysis of the code would not be sufficient to detect all the possible behaviors. The behavioral analysis method consists in observing a file while it is being executed and recording its behavior. This method is used to extract behavioral features. Some examples of these features are accessed files, network connections and created mutex. The goal with these features is to exploit the idea that authors tend to reuse code and have certain preferences concerning the behavior of their software.

- Resources-based features. Another difference is that modern programs

use internal and external resources. When source code is compiled to produce executable code external libraries are linked to the original code. Also other resources such as icons, library code and images are integrated in the binary, i.e., PE files and .apk files contains libraries, icons, images and other resources. As a result when binary code is executed, these external resources might be requested and used on the fly. Examples of these resources include URLs visited by a software without user interaction, images or sounds embedded in the executable files, libraries and compiler's versions used to create the executable files. These features could be extracted during static or dynamic analysis. The goal with these features is to exploit the idea that authors reuse external resources for their software. This last level of features has received very little attention from the research community.

Feature extraction and feature selection are very important steps in author attribution analysis. For better results it is common to use a mixture of features. One option is to embed the features for future processing, or create multi dimensional representations for the analysis.

2.1.2 Representations

Extracted features can be represented in a form suitable for automated analysis. It should also be noted that combining features into a unified form may result in the establishment of new software metrics. Different representations

Table 2.1: Summary of features and some examples.

Lexical	token frequency length of the lines number of operands number of operators number of spaces frequency of words
Syntactic	number of functions number of reserved words number of comments in a single line number of semi-colon (;) number of classes existence of comments class names method names text strings contained
Semantic	number of loops control flow graphs functionality analysis cyclomatix complexity
Behavioral	accessed files network connections created files created mutex encryption algorithms used
Resources-based	libraries included tool-chain used files retrieved icons or images embedded urls included

of features are possible: from a very efficient form but not expressive enough to retain useful information, to a very expressive form but computationally inefficient. Then we must select a proper way to represent the features since different representations could help in different ways the methods for authorship attribution. The most commonly used representations are: token, string, n-gram, idiom, vectors, sets, trees, graphs, embedding.
The following paragraphs will describe them with details.

Token. A token represents by itself the extracted feature. This token could be a word, a character, symbol, function, basic block, etc. Usually we combine the tokens representation as a binary feature or by their frequency. This representation does not capture any semantics of the code, instead it focuses only on the tokens. In work [87], the authors measure the existence of

specific tokens for the Pascal programming language to perform authorship attribution. Some of the tokens they measure are: inline comments, blocked comments, keywords followed by a comment, indentation, underscore used in identifiers, "BEGIN" keyword followed by a statement on the same line, multiple statements per line, and blank lines in the program body. In work [65], the authors proposed to measure specific tokens such as: number of distinct operators, number of distinct operands, total operator occurrences, total operand occurrences. These numbers are used to create a profile for the authors.

Strings. A string describes a sequence of tokens. It could represent a complete program or only part of it. In work [103], the authors present Jplag to detect plagiarism in Java source code. They first convert the source code to a string of canonical tokens, then they use the string tiling algorithm to match similar content on the source code. In work [75], the authors create a sequence of tokens from the Java class file, analyzing the byte code and not the source code. Then they evaluate the similarity between token sequences using the adaptive local alignment, finding relevant parts where the code is the same.

N-gram. A n-gram combines n terms together to express more information. Different from tokens and strings, n-gram helps to combine the contextual and the neighboring information. In work [19], the authors use n-grams to represent the source code. They construct a system to query authors based

on the documents, so they can rank and match how similar the evaluated work is with the profiles.

Idiom. An idiom is a short sequence of instructions, (commonly between 3 and 6 instructions), with optional single-instruction wildcard. This wildcard offers more flexibility to compare sequences of code into patterns, instead of strict sequences which can be altered by obfuscation. In work [112], the authors use idioms to capture patterns indicative of compiler provenance among other techniques.

Vectors. A vector is an ordered list or tuple of a fixed number of elements or dimensions. A feature vector describes the frequency of particular features. Vectors are one of the simplest representations and are efficient to work with. If the number of features is very large, then we should apply dimensionality reduction. In work [107], the authors use relevant n-grams for a class-wide document frequency of executable programs to detect computer viruses. They combined several classifiers to improve the accuracy, representing their training samples as vectors.

Sets. A set is a collection of unique objects in which the order of its elements has no significance. One can convert vectors to sets, or even use sets of vectors. In work [3], the authors create a set of APIs from each analyzed Android app. Then they conduct a frequency analysis to select the ones that were more frequent on the malware samples than in the benign samples.

Authors use this set of features to classify malware.

Trees. A tree is an undirected acyclic graph which can be used to represent the structure of data. They are less general than graphs. Abstract syntax trees and parse trees are examples of this representation. In work [32], the authors employed Bayesian classification of hierarchical features for the JavaScript abstract syntax tree to identify syntax elements for probable malware.

Graphs. A graph represents a model structure in the data. It express the semantics of a software. One could construct it directly from the source code. From binary code it is necessary to decompile first to source code, or intermediate representation code and create from there. Another option is to instrument the binary to produce it from the callable path of instructions. Some program features naturally represented as graphs include control flow graph, call graphs, dependency graphs, graphlets. In work [111], the authors use control flow graphs and graphlets to recover tool chain provenance. They use machine learning techniques to classify software based on the tool-chain used to produce it.

Embeddings. One representation of features may be embedded in different representation for efficiency, tractability, speed, storage, etc. Strings may be embedded inside sets or vectors. N-grams and trees may be embedded inside vectors. Graphs may be divided into subgraphs, embedded inside strings

and then embedded again inside sets of vectors. In work [51], the authors used embedded function call graphs to detect Android malware using machine learning techniques.

It is always possible to use a mixture of the representations. All these representations contain different levels of information from the extracted features from lexical to semantic information. Chosen representations will have an impact on the accuracy of the results. They also will impact time and computer resources required to extract, represent, process and perform the analysis.

2.1.3 Authorship attribution methods

After extracting and representing features in a desired format, the next step is to employ some method to attribute a work or program to a candidate author. An authorship attribution model should be constructed based on the previous known work from the candidate authors, then used it to attribute new work to a candidate author.

Authorship attribution analysis could be addressed from two different perspectives. From a closed-world problem perspective, we can only attribute new work to a previous known author, we can not see outside of our known author profiles. From an open-world problem perspective, we need to discover and classify new authors, no matters if we previously see them or not. Usually this approach had been used more to cluster or group similar authors than to attribute new author works.

Also it is important to note that authorship attribution analysis can be seen as a classification problem, where the classification process plays a critical role here. As a classification problem, we can use a set of works or codes from a known author to create and train our classification model, then this model is used to classify a new sample.
In addressing authorship attribution analysis, one can pursue one or several of the following results:

- Authorship Identification. From a set of candidate authors we will match the most probable one that produces a specific work under study.
- Authorship Clustering. From a set of candidate authors we will match the ones that share stylistic similarities, for example programming skills.
- Authorship Evolution. From a set of works from the same author we will infer the evolution of programming skills or development preferences.
- Authorship Profiling. From a set of candidate authors we will extract specific profiles like gender or nationality, based only on the stylistic features extracted from their work.

As a classification problem, the following approaches could be employed : profile-based, instance-based or hybrid approaches. In the following paragraphs more detail on different approaches are given.

2.1.3.1 Profile-based approach

In a profile-based approach an author is represented through a set of characteristics accumulated from concatenation of all his works, i.e., programs.

Probabilistic Models These methods attempt to decide on the probability $P(x|a)$ for a code x belonging to a candidate author a. The attribution models find the author more similar to the set of existent profiles. The profile is created with a selected set of features from all the known codes from an author. Well known algorithms used in this model are the standard naive Bayes classifier, Markov chains, Hidden Markov chains.

Compression Models These methods attempt to calculate the cross-entropy between the known code and the unseen code. It does not produce a concrete vector of features to represent author profile, but just a reference between the original size of the representation features before and after compression. First, all the features' representations for an author a are added and compressed $C(x_a)$. Then one adds the unseen code x to each of this x_a's and compressed $C(x_a + x)$. The difference $d(x, x_a) = C(x_a + x) - C(x_a)$ indicates the similarity between the profile and the unseen code. Finally, it calculates all the differences to find the closer one.

Common n-grams This method is particularly interesting and has been studied for several years. It represents the author's profile with the L most frequent n-grams representation from the source code or the binary code.

Then a distance metric is employed to find the similarity between the profile and the unseen code. The typical distance to calculate the similarity between two programs x and y is:

$$d(P(x), P(y)) = \sum_{g \in P(x) \cup P(y)} \left(\frac{2(f_x(g) - f_y(g))}{f_x(g) + f_y(g)} \right)^2 \tag{2.1}$$

where $P(x)$ stands for the set of L most frequent n-grams in x, and g is the n-gram while $f_x(g)$ and $f_y(g)$ are the frequencies of occurrences of that n-gram in x and y, respectively. This method has two important parameters that must be addressed: the length L and the number n for the n-gram. The study [48] reports promising results on author attribution analysis with this method.

2.1.3.2 Instance-based approach

Where each author's piece of work is considered as an independent unit when training a model, the problem now is to find the most similar works to an unseen code. The authors of these works are the probable candidates for that code. A slight variation of this model is to divide the complete code into smaller pieces as functions of basic blocks for example.

Vector Space Models The features could be represented in many diverse ways, so we could consider each code as a vector in a multivariate space. Diverse machine learning algorithms can be used to build a classification model,

including Support Vector Machines, principal component analysis, decision trees, neural networks, genetic algorithms, and classifier ensemble methods. These algorithms have been studied in the authorship attribution problem and in other fields. Usually these methods can handle high dimension, noisy, and sparse data, allowing more expressive representation of the codes.

Similarity-based Models This model calculates the pairwise similarity measures between the unseen code and all the training codes; then it estimates the most likely author based on a nearest-neighbour algorithm. The pairwise similarity can be calculated using other methods including the compression model or the n-grams model. Another alternative is to use edit distance to measure the dissimilarity.

Meta-learning Models Not only the general purpose classification algorithms might be used in the authorship attribution problem. One could create more complex algorithms specifically designed for this task. Alternatively, one could create an ensemble of several methods to improve the accuracy.

2.1.3.3 Hybrid approach

This approach borrows some elements from the profile-based and instance-based approaches. For example, one can do the training over known instances, and then using the results to create a profile from the averages of the features' representations.

A combination of methods could be used to improve the accuracy of the results. Depending on the final goals one method could provide better results than others. Each method has its own strengths and weakness.

2.2 Basic introduction to Android platform

The Android platform is a core component to several devices used these days, from mobile phones to smart appliances. Using applications or apps is a way to expand the use of those devices. These applications comprise from social media apps to games, utilities, office production apps and more could be found in specialized virtual stores called "Markets". GooglePlay is the official market for Android apps, but is not the only one. Several other markets are available around the globe, known as third-party markets, they offer their unique contribution to the Android ecosystem. F-droid [126] market offers only open source apps which their source code is freely available. Yandex [136] market offer apps in Russian, targeting people that speak that language. Other markets target Chinese language speakers, or are specialized on games. Even other markets offer pirated games or popular paid apps for free, most of them with adware or malware added. It is possible to find apps directly accessible on Internet, without any market intervention.

With all those options, users can search and install apps from diverse markets, exposing their mobile phones to be victims of malware, adware or ransomware.

Official GooglePlay market is the most recommended one to install apps, because they have some security measures in place. However, some malicious apps have been discovered in recent years in this market.

Android apps An Android app is written in the Java language and compiled into a .dex file which is the main executable part. The apps are packaged into an .apk file as showed in Figure 2.2, each app contains the executable .dex file; the AndroidManifest.xml file that describes the content of the package including the permissions information; native code (optional) in the form of an executable or libraries that are usually called from the .dex file; a file with a digital certificate authenticating an author; and the resources that an app uses (e.g., image files, sounds). Each .apk file is annotated with additional information, so called meta-data, such as the app creation date and time, version, size, etc.

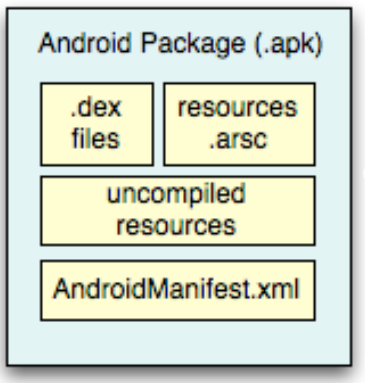

Figure 2.2: Android package (.apk)

Digital certification plays an important role in Android apps. This is the

only mechanism developed to attest the ownership of an app. The certificates are self-signed, i.e., no certificate authority (CA) is required. While the mechanism was originally meant to tie an author to an app, allowing a legitimate owner to issue new apps and update older versions under the same key; several problems have quickly surfaced. Through a preliminary analysis of Android markets, we discovered that the authors (both legitimate and malicious) tend to generate a new pair of private/public keys for each application. The value of the self-certification is also undermined through an extensive use of public keys made available for debugging purposes. Although these days the official Google play market bans apps signed with these keys, other markets do not seem to enforce this policy. Finally, the recently discovered master key vulnerability opened a new window allowing attackers to inject malware into legitimate apps without invalidating a digital certification [46]. These weaknesses challenge the legitimacy of using digital certificates for app authorship identification. In this light some of the methods designed to rely on the existence of the original app (determined based on the certification) for detection of plagiarized applications (e.g., [142]) require readjustment.

Developing Android apps An author of an Android app must have some basic tools to produce her app. These days AndroidStudio is the latest Integrated Development Environment (IDE) to create Android apps, replacing Eclipse-plugin in 2015. Within this IDE a developer will also need the Android Development Toolkit (ADT) which includes basic tools to produce

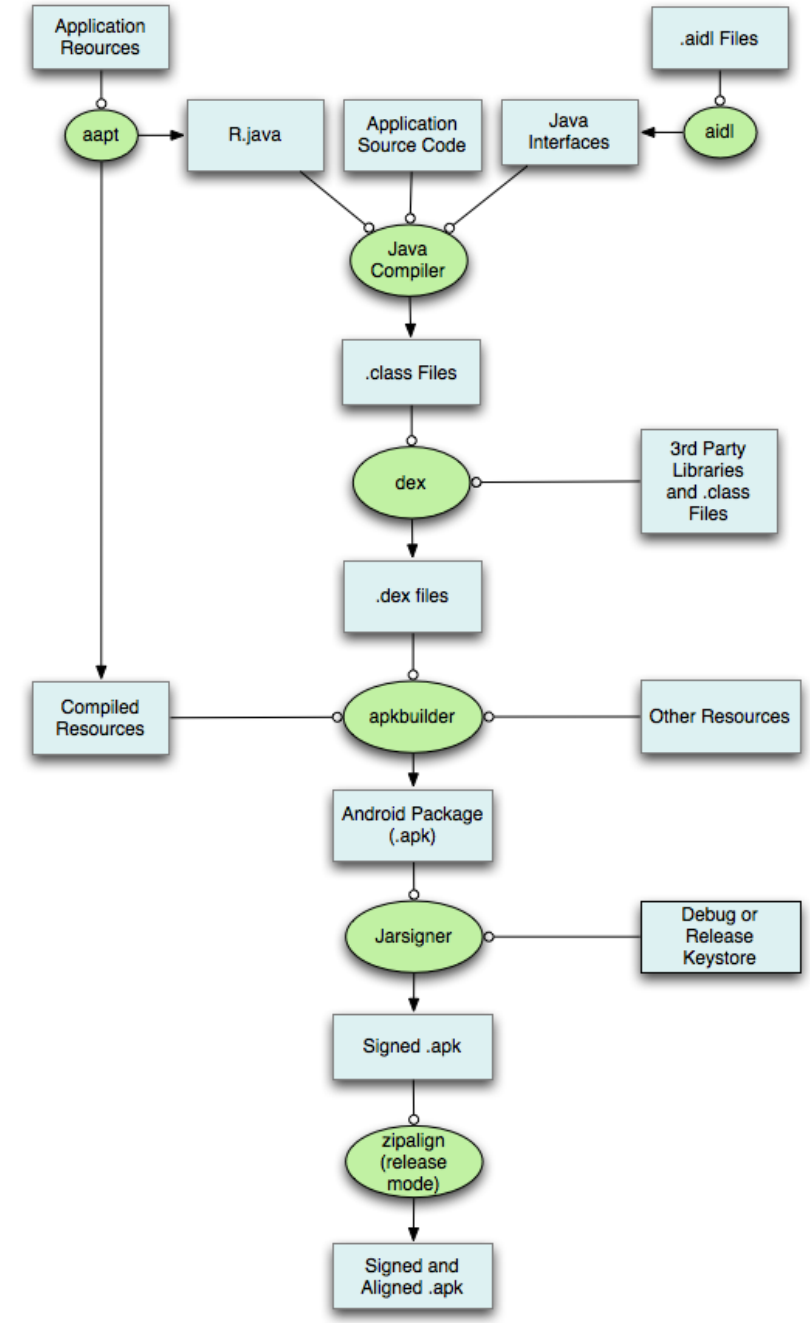

Figure 2.3: Android application project build process. [125]

Android apps. The Java Development Kit (JDK) is also necessary to execute AndroidStudio and to compile Java language code into .jar files.

An Android application project build process consists in seven steps: Resource code generation, interface code generation, Java compilation, byte code conversion, packaging, signing, and package optimization. An official diagram of this process is showed in Figure 2.3. Green shapes show tools from ADT, except from the Java compiler and jarsigner, which are part of JDK.

Is important to note that the core of an .apk file is a .dex file, wich contains executable code for the dalvik virtual machine (DVM). Recently ART runtime was introduced to improve the performance of Android apps, this ART runtime is compatible with DVM, but instead of being translated each time, the code is compiled from a .dex file at installation time.

To produce this .dex file, Java source code is compiled into .class files, then into a .jar file. At this stage, it is possible to use a tool to optimize and/or protect .dex files, like proguard which is provided with the Android Development Toolkit (ADT), or other commercial products that work at the .jar level. Finally a .jar file is transformed into one or more .dex files.

Besides the official tool set, a developer can use other languages and tool sets to create Android apps, such as C# with Dot42 or Xamarin; Basic with Basic4Android; lua with coronaSDK; HTML5 and javascript with Titanium and Phonegap; ruby with roboto and rubymotion; swift with swift for android; and even without programming at all a developer can use gamesalad

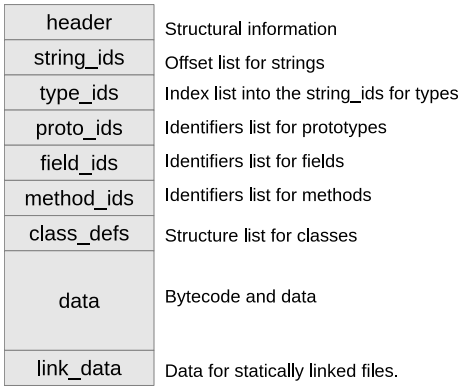

Figure 2.4: The structure of a typical plain .dex file.

to create an Android app.

Other tools to protect apps from being reverse engineered could encrypt the original .dex code and replace it with a loader, that uses native code to decrypt the original code. Some examples of this packers are ijiami and baiduprotect.

Executing Android apps An Android app installed on a device contains one or several entry points on the AndroidManifest.xml file. These entry points could be present in the form of a "Main Activity", "Service", "Provider" or even an "intent" which is called when a configuration change occurs. All of those entry points make reference to one or different classes, there is nothing like a single "Main" function or principal Class directly in the code, different from other programming practices. From a .dex file it is possible to call native functions through JNI interfaces, or even call other .dex or .jar files, but the main important file to analyze is classes.dex.

The classes.dex file structure is documented in [9] with a specification format, indicating how the file is organized, showing that almost everything is about references and which are the constrains for the format. This file commonly embeds libraries and support code originally in .jar format, making files self contained, but at the same time, much binary code is duplicated on a device running several Android apps. This is also an advantage as each app should be isolated from the rest non related apps, offering a certain level of security. The structure of a .dex file is shown in Figure 2.4. This file is partitioned into a number of sections that describe aspects of a .dex file. The header contains offsets and sizes of other elements. Identifier lists are comprised of offsets pointing to corresponding entries usually located in the data section. Dalvik specification is very clear in the first sections, and explicitly says that the offsets of elements should be located in the data section. Some references contain more offsets where the names, types and code are located, i.e. Class references. Strings defined in the code and used by the app are located in a specific section in a .dex file, they should be organized alphabetically. Class definitions are located in a different section, and actual method code are in a different place.

2.3 Related work

The problem of program/software authorship attribution is not new. As we introduced the authorship attribution process in Section 2.1, here we offer a

brief overview of the related work from different perspectives, Section 2.3.1 presents the first works related to source code authorship attribution, and its evolution to current applications as plagiarism detection, copyright infringement, and code cloning. However, due to the lack of source code, specially in the malware domain, these source code authorship attribution analyses have limited applicability in malware analysis. This opportunity opens interests in binary code authorship attribution. Section 2.3.2 shows works related with binary code authorship attribution in general. Section 2.3.3 shows studies related with malware or binary analysis, that provide us with features or ideas, but they are not related with Android neither authorship attribution. Finally section 2.3.4 shows works related directly with Android features, all these works are not related with authorship attribution, but present useful features usually employed to do malware classification.

The evolution of research in the plagiarism detection and authorship attribution areas are shown in Figure 2.5. Circles represent the initial studies on software authorship attribution. Squares represent plagiarism detection studies. Triangles represent studies on software authorship attribution.

2.3.1 Source code authorship attribution

One of the initial goals of authorship attribution in the software domain was to understand how to measure algorithms and their complexity. In the 70's the theory of "Software Science" was introduced [18, 65]. The authors tried to apply science laws to source code, i.e., measure how pure is an algorithm

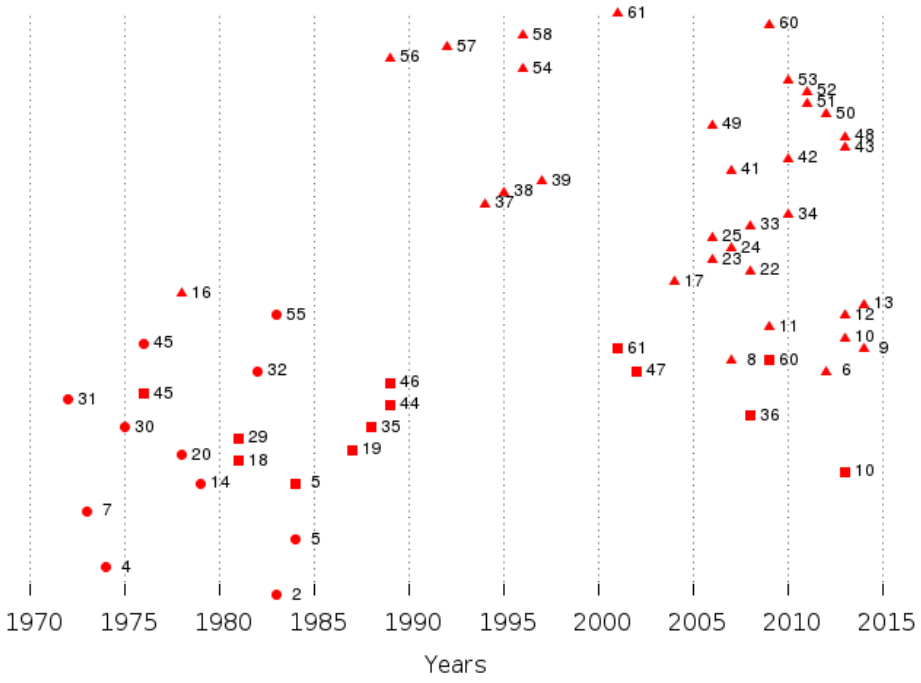

Figure 2.5: Evolution of studies on software authorship attribution. The numbers are related with the bibliography in Appendix A.

based on the relations between operand and operators. Over the years this theory was widely used [13, 66, 98, 31, 4], validated [45], criticized [120], and even disproved [67]. Nevertheless, it set a precedence in software metrics and estimations. It was not until 1984 when Berghell [15] demonstrated that the metrics proposed by this theory are not effective in detecting plagiarism and in general, they are not useful measures for similarity comparisons in modern languages.

These research efforts led to the exploration of how to automatically evaluate algorithms in assignments in programming courses. There were several at-

tempts to measure the effort of students in programming courses with little success [97].

Naturally, evaluation of assignments evolved into detection of plagiarism. Plagiarism detection was always centered around popular programming languages, i.e., FORTRAN [98, 15, 39] in the early 70's; Pascal in the early 80's [64, 43, 99, 74]; and Java in 2000's [75, 134]. In all these studies the primary focus was on analyzing the source code.

After the first Internet Worm analysis [121], Spafford and Weeber introduced in 1992 the term **Software forensics** in their study [122]. In their study they drew links between traditional law enforcement investigation and investigation of computer system breaches, proposing for the first time to link the remains of source code found in the computer server after the attack to an specific author. Some drawbacks for the proposed theory to link code to an author were shown by Krsul [84], such as: people reuse code, team work, and programs are altered by code formatters and pretty printers. As a result from these studies, in 1995 Krul and Spafford proposed to enhance real-time intrusion detection systems by including authorship information [85].

Authorship attribution feasibility has been shown in a small pilot study by Krsul et al. [86] and has since revolved around the idea of attributing source code through various characteristics of a programmer style [70].

Over the years, several approaches to profile authors have been proposed. Structure-metrics versus attribute-counting systems were reviewed by [127], where the former performs equal or better than the latter system. Statisti-

cal methods for program profiles were proposed by [114]. Krsul et al. [87] introduced the idea that **tools or libraries can influence the stylistic features of an author**.

Krsul et al. suggested that it is possible to find an author in a predetermined set of authors [86]. This result, however, was later disproved by Ding [38]. Ding applied the same metrics to the Java programming language with not satisfactory results, concluding that **programming language plays a significant role in the selection of the appropriate metrics**.

More studies on profiling authors based on statistical features and n-grams at source code followed [49, 50, 48, 19]. The source code author profile (SCAP) based on byte-level n-grams were presented in [48] as a promising approach, and some improvements related with the length of the profile have been done by [47, 92]. Later on, Layton et. al [93] proposed an approach to perform automated unsupervised authorship analysis from source code using hierarchical clustering.

A different approach using stylistic counter-metric features was developed by Lange and Mancoridis [90], here the authors first used normalized histograms to create author's profiles, then they used a genetic algorithm's search to reduce the search space in the classifier.

The *Consistent Programmer Hypothesis* [71] is another approach used as ground for the authorship attribution problem. This hypothesis states that the programmers are "consistent and competent", it then extended the *Competent Programmer Hypothesis* [35] which postulates that: "The program-

mers create programs that are close to being correct!". This consistency hypothesis was the basis for other studies in authorship attribution of source code [71, 70].

One of the largest focuses in source code authorship attribution is plagiarism detection [124, 21].

In recent study by Burrows et al. [20], authors compared the published techniques for authorship attribution of source code, showing that information retrieval approaches with n-grams from [19, 48] are the best techniques in source code authorship attribution. The first technique proposes n-grams at token level and the Okapi BM25 similarity metric in a ranking approach. The latter one proposes the SCAP method, which uses n-grams at byte level and the frequency of them to create an author profile.

Only a few studies on authorship attribution had been performed on malware source code. Layton and Azad [91] perform an authorship attribution analysis of the leaked source code of the infamous zeus botnet, they found that this malware was written by more than one person. Since there is no real information on this malware authors, results from this study are theoretical and have not been corroborated yet.

2.3.2 Binary code authorship attribution

A key point in the theoretical study by Walenstein [129] was that the source code is not always available, especially in the malware domain. In this case other methods should be used to extract features from binary code, such as

studying the behavior of the software [131], or finding software birthmarks in the programs. A **software birthmark** can be defined as the inherent characteristic of a program extracted from the program itself. Some studies used software birthmarks based on API calls [25, 23] or control flow graphs [134] for similarity detection. In a study by Chae et al. [23], authors proposed to use cosine similarity over a vector of API frequency calls to analyze software similarity. Other methods of analysis at the binary code level using n-grams were also proposed by Reddy et al. [107]. Although these previously mentioned studies did not aim specifically for authorship attribution, these features can be explored for authors profile creation and authorship attribution.

The problem of binary code authorship attribution has been explored by Rosenblum et al. in a series of studies [112, 110] that investigated toolchain provenance as part of authorship attribution. Rosemblum et al. [110] proposed the analysis of binary code to do authorship attribution using machine learning techniques and automatic feature selection. Previously the same authors investigated the compiler provenance [112] from binary programs, where they found it feasible to automatically discovery the details that characterize the production process of a given binary (e.g., the compiler family, versions, optimization options and source languages). This proposed approach was later extended by Alrabaee et al. [6], authors employed a multilayer method integrating syntax-based and semantics-based features to perform authorship attribution over windows binaries.

Another proposal for forensic analysis was to use dead-codes and other programming blunders when analyzing binary codes [16].

The ideas of authorship attribution can be applied to discover different levels of the software origin, as mentioned before such as compiler and tool-chain provenance, or metamorphic engine attribution for malware. Chouchane et. al [26] demonstrated the feasibility of attributing the metamorphic engine that created the malware under review. The authors proposed to use n-grams to undercover the traces of metamorphic engines obtaining good accuracy with all but only one of the metamorphic engines studied.

All these works focused on identifying the tools used to create the binaries, as a first step to detect maliciousness or engine attribution. Although all these results showed high accuracy of correct attribution among closed world problem over well defined engines/tool-chain components, the question of applicability of these results to extracting developers' fingerprints on the wild remains.

2.3.3 Malware analysis

Malware analysis has been studied for many years now, some studies that condensed this field [40, 72, 132, 68] show us evolution of different techniques, methods and focus on this research area. The importance of previous works in malware analysis for our study is to learn about features, feature extraction, classification, and clearly understand the differences between malware classification and authorship attribution.

In malware analysis, the main focus is to analyze binaries files, i.e., behaviour of the software [130], API calls [23], and control flow [82]. This focus on analyzing binary files is also in our interest. Other work studied the binary structure, representing it as images [27] to find similarities between samples in the same malware family. New ideas on how to extract and use features are inspired from these study.

Although some studies focus on performing a binary classification, i.e.,malware vs. non-malware; other studies focus on malware family classification, assigning one class per studied family. Rieck et al [108] discuss the inefficacy of signature based approaches for detecting malware, proposing an approach based on extract behavior patterns, embedding them into a feature vector and use support vector machines method as a classifier. The authors experiment with benign software and malware different from the families used in the training phase, discarding this unknown samples from the system. Bailey et al [11] proposed a behavior-based clustering technique to classify malware families. The authors created a set of features from extracted behavior, then they employed a hierarchical clustering algorithm whose distance function is normalized compression distance. Kinable and Kostakis [81] extract call graphs from malware samples and experiment with k-medoids and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) techniques. This approach offers to discover new variants of already studied malware.

Mekky et al [95] classified malware families using features extracted from network traces. The authors employed the random forest classifier to identify

between malware network traffic and noise network traffic produced by other applications. They employ the same classifier to label known malware families. Rafique et al [105] focused on classifying malware families by comparing evolutionary algorithms based on genetic algorithms versus typical machine learning algorithms.

Although the primary focus of these approaches was the extraction of "software birthmark", i.e., a combination of unique malware characteristics, the results of these studies can be complementary for authorship attribution and should be further explored in this context.

2.3.4 Android malware analysis

The past decade has been marked with extensive research in the area of mobile security. A broad study looking at a variety of mobile technologies (e.g., GSM, Bluetooth), their vulnerabilities, attacks, and the corresponding detection approaches was conducted by Polla et al. [88]. More focused studies surveying characteristics of mobile malware were offered by Alzahrani et al. [7], Felt et al. [44] and Zhou et al. [143].

With the recent burst of research interest in the area of Android device security, there have been a number of studies focusing on mobile malware detection. These studies include detection of privacy violations during apps' runtime (TaintDroid [41], MockDroid [14], VetDroid [140]), unsafe permission usage [42, 12, 115, 37, 118] and security policy violations [96, 116].

There were a number of general studies offering methods for malicious app

detection. These methods can be broadly divided into those focused on the detection prior to app installation (e.g., market analysis) and those that monitor app behavior directly on a mobile device.

Among the studies in the first group are RiskRanger [63], DroidRanger [144], and DroidScope [135] that dynamically monitor mobile apps behavior in an isolated environment collecting detailed information that might indicate maliciousness of a sample. Similarly, DroidMat [133] and DroidAPIMiner [3] looked at identification of malicious apps using machine leaning techniques. Since these techniques are computationally expensive for a resource-constraint environment of a mobile platform, they are mostly intended for an offline detection. A number of studies introduced lightweight approaches to malware detection to be applied on a mobile device directly, among them is Drebin [10]. This approach employs static analysis in combination with machine learning techniques (SVM) to detect suspicious patterns in apps through signatures.

With a burst wave of repackaged applications, a number of studies looked at the problem of mobile apps similarity. The majority of the existing methods look at the content of .dex files for app comparison. Juxtapp [69] evaluates code similarity based k-grams opcodes extracted from selected packages of the disassembled .dex file. The generated k-grams are hashed before they undergo a clustering procedure that groups similar apps together. Similarly, DroidMOSS [142] evaluates app similarity based on fuzzy hashes constructed based on a series of opcodes. DroidKin [62] detects the similarity between

apps based on analysis of relations between apps. ViewDroid [139] leverages user interface-based birthmark for detecting app repackaging on the Android platform. Several methods were developed to fingerprint mobile apps to facilitate the detection. For example, AnDarwin [29], DNADroid [30], and PiggyApp [141] employ Program Dependance Graphs (PDG) that represent dependencies between code statements/packages. Potharaju et al. [102] computes fingerprints using Abstract Syntax Trees (AST). All the studies require a pair-wise apps' analysis for detection of app similarity/repackaging.

Another work look at the complexity metrics on the Android app [104], the authors combined these metrics with app permissions and found an acceptable accuracy for malware detection. Extracting complexity metrics is computationally expensive though.

The PlayDrone study by Viennot et al. [128], focused on the characteristics of GooglePlay market, the authors looked at the libraries' usage. In a similar vein, AdRob [52] looks at library usage and the popularity of markets among cloned applications.

Several other studies employed machine learning techniques to classify Android malware. Peiravian and Zhu [101] employed API calls and permissions with SVM, decision trees (J48) and bagging predictors. The SVM method outperformed the rest. Amos et al. [8] employed 256 features extracted from dynamic analysis of Android malware, and tested them with Random Forest, Naive Bayes, Multilayer Perceptron, Bayes net, Logistic regression and decision trees (J48). Random Forest outperformed the rest in Cross valida-

tion, but came second after Bayes net in Testing set. For Shabtai et al. [119], Random Forest outperformed the rest classifiers when comparing Games versus Tools apps. Sahs and Khan [113] extracted permissions and control flow graphs to train One-Class SVM. The authors applyied reductions to the graph to address polymorphism. They also described three kernels used to train their approach, a Kernel over binary vectors, a kernel over strings and a kernel over graphs. The results from this study showed high values for miss-classified legitimate Android apps.

Between the features used when analysing Android apps, Dalvik bytecodes is one of the more popular [80, 22, 5, 79, 137, 51, 77].

2.4 Concluding remarks

In this chapter we covered a basic introduction to the authorship attribution process, which is fundamental for a better understanding this thesis work. We also cover a basic introduction to Android apps and their building process. This basic knowledge will be referred to later in Chapter 3 when explaining in detail our proposed approach to perform Android apps authorship attribution.

Beyond the academic world, a number of other methods were employed to enrich the authorship attribution analysis, such as collecting intelligence from malware campaigns, collecting information from external resources employed and its historical records, using social engineering and offensive security to

unmask the persona behind the attacks. Examples of these activities can be found in the following public reports [94, 106, 28] that investigate a specific group behind massive malware campaigns. Following an analysis of the malware samples, the researchers extracted external features and traced names and addresses associated with these samples, finding specific people involved with this specialized group. Collecting personal information available on the Internet, the researchers took a step further creating attackers' profiles and their affiliations. It should be noted though that in spite of the successful outcome, the methods employed were labor intensive, manual, and often questionable from an ethical perspective.

The reviewed related work shows efforts in authorship attribution over the source code domain. A few studies have been done targeting authorship attribution on binary code, showing that is difficult but possible to perform this task. In spite of the vast research efforts in this area, there are still some challenges remaining:

a) Closed world problem: It is not possible yet to attribute a work for unknown authors. Techniques and models focus on a set of known authors to perform attribution. To resolve this we designed a model for automatically discovering new author profiles and incrementally expanding the catalog of known author's profiles.

b) Lack of source code: The source code is rarely available for malicious software. Even though it is possible to reverse engineer or decompile a malware sample, the results are far different from the original source code because of

the transformations applied during compilation and decompilation. In our work, we focus on binary code.

c) Lack of datasets: Recently a few studies in the Android malware analysis domain offered their data to the research community. Unfortunately, these datasets are not appropriate for authorship attribution as no author information is available. We created two datasets for Android authorship attribution purposes, one for legitimate authors and the second one for Android malicious authors. These datasets are available to the academic community.

Chapter 3

An automatic authorship attribution technique for Android applications

"If we knew what it was we were doing, it would not be called research, would it?"

Albert Einstein

In this chapter we present an overview of proposed approach. We describe each phase in detail later. Finally, we present technical information about the prototype developed.

The goal of the proposed approach is to systematically attribute Android apps to the corresponding author's profiles. Typically, attribution studies in the literacy domain focus on identifying an author out of a sample from a

set of candidates, based on the sample's similarity to stylistic discriminators found in reference profiles. In the malware domain, this method has a limited value as reference profiles only represent a small set of known authors. As such, it is necessary to step beyond the known set and group stylistically similar binaries not attributed to a existing profile as they might potentially be linked to a new author.

Discovering new profiles in a stream of apks is a problem not studied before in authorship attribution. With our approach we are able to address authorship attribution as an open world problem and it includes novelty detection techniques to create new author profiles over time.

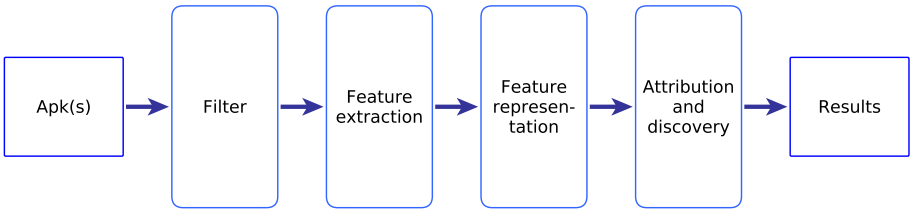

Figure 3.1: The flow of proposed approach to work on Android apps authorship attribution.

An overview of the flow to our proposed approach is showed in Figure 3.1. This proposed approach takes as an input an Android app. Then the approach proceeds with the *Filter phase*, necessary to discard packed and non-valid apps from further singular analysis. In the *Feature extraction phase* the .dex file is extracted and parsed to retrieve several sets of features namely features that are extracted directly, features inferred from the strings sec-

tion, features composed by opcodes, features composed from the .dex file and features not related with the .dex file. It is important to notice that all these chosen features are related to the author's decisions, or toolchains employed. The *Feature representation phase* is separated by convenience, we can perform this job in a distributed environment. In this phase we combine all extracted sets of features in a single feature vector. Finally, in the *Attribution and discovery phase* we employ the proposed hybrid meta-learning method, capable of incremental discovery of new author candidate profiles. Several novelty detection algorithms include offline and online training [34], our approach is similarly composed of two steps: reference profiles creation (offline) and discovery of emerging profiles (online).

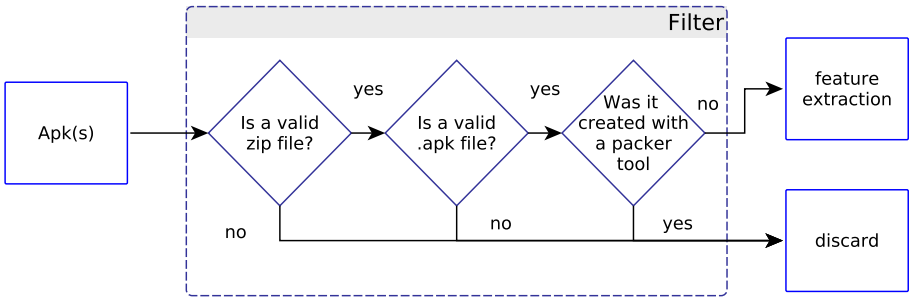

Figure 3.2: Overview of filter phase.

3.1 Filter

An Android app in the form of an .apk file is the basic input for our framework. We need to ensure that each file is a valid source, e.g., the input

file is a known file that our system can handle, it is not corrupted, contains valid information. It is important to note that an .apk file is a compressed zip file that should contain AndroidManifest.xml, the classes.dex binary file, certification information and other resources (Figure 2.2 shows Android app basic elements). In the *Filter phase* we validate an input file with three steps shown in Figure 3.2. In the first step we ensure that the input is a valid zip file, if it is not, it is discarded. Then in the second step we confirm that it is a valid .apk file containing AndroidManifest.xml and classes.dex files, these files are important for use because they are the main sources for our feature extraction phase. These files are also necessary to execute an app in a mobile device. An input file is discarded if this second test fails. Finally in the third step we employ a signature-based approach to detect if an input file was created using a known Android packer tool. Packing, an app protection mechanism would interfere with our analysis. Android packer tools are employed to add a security layer to an original app. Packer tools use encryption to protect code and resources, replacing the .dex file with their own loader and decrypter code, resulting in similar .dex files even when the apps are completely different and unrelated. In this case our framework analyzes the generic code, not the original one created by an author and his tools. In this case, results produced are not useful.

Accurate detection of obfuscators or packers is a challenging problem. For our purpose we build a signature-based detection engine. This method is inspired by Yu [138], which depends on the presence of folders and files used

Table 3.1: Android packers detected.

Packer Name	Indicative files and folders present in packed apps.
Bangcle	Folder *bangcleplugin* contain .dex files. Android libraries *libsecmain.so* and *libsecexe.so* are also present.
360	Original .dex file is saved as *libjiagu.so* or *libjiagu_art.so*.
Baidu	Files *libbaiduprotect.so*, *assets/baiduprotect.jar*, *lib/armeabi/libbaiduprotect.so*, and *assets/libbaiduprotect_x86.so* are present.
Tencent	Libraries *libmain.so* and *libshell.so* are present.
Ijiami	Libraries *libexecmain.so*, *libexec.so* and file *ijiami.ajm* are present.

by a known packer. A list of known Android packers which our framework can detect is showed in Table 3.1. These signatures to detect packer tools were manually crafted after apps analysis, therefore new signatures should be created for new packers tools.

During this phase general information and meta-data from an app are extracted and stored for further usage.

3.2 Feature extraction

Since Android apps source code is not always available, the majority of the features used or proposed for source code analysis by other studies are not applicable. In general it is possible to transform binary code to the corresponding version of source code through reverse engineering. This however results in generic source code affected by machine transformations produced

by tools used to encode and decode; losing most of the useful values for source code analysis; e.g., spaces or variable names.

Feature engineering is a critical step in machine learning as experimental results are driven by chosen features. In this work, we employ features directly influenced by authors' preferences. A total of 40 features with different lengths and formats are extracted, and then transformed and combined through feature representation into a *feature vector*. These features enable us to accurately perform authorship attribution using machine learning techniques.

Table 3.2: Groups of features.

Group	Sets of features
features that are extracted directly	14
features inferred from strings section	8
features composed by opcodes	4
features composed from .dex file	4
features not related with .dex file	10

Spafford and Webber [122] set precedence with their "software forensics" work, suggesting useful features that can be extracted from binary files. This features include:

- Data structures and algorithms.
- Compiler and system information.
- Programming skills and system knowledge.
- Choice of system calls.

- Errors.

They also suggested other features that can be extracted from source code files.

- Language.
- Formatting.
- Special features.
- Comment styles.
- Variable names.
- Spelling and grammar.
- Use of language features.
- Scoping.
- Execution paths.
- Bugs.
- Metrics.

Our proposed working features are extracted mainly from .dex files of an Android app. These files are binary bytecode containers of code for the app. We selected values, software metrics, text strings, and other characteristics through careful analysis on Android apps content and the process of building apps. From all these possible features extracted from an app, we chose those which were influenced by **an author's preference**; for example data structures, class names and code employed. Table 3.2 shows five groups that contain our set of 40 features. To extract most of this information we need to analyze a .dex file from different points of view, as a result, several .dex file parser software were developed. More details of these .dex file parsers at Section 3.5.3.

Table 3.3: Directly extracted features.

#	Feature
1	Total number or classes
2	Total number of classes containing interfaces
3	Total number of classes containing information about source file
4	Total number of classes containing annotations
5	Total number of classes containing data
6	Total number of direct methods
7	Total number of virtual methods
8	Total number of abstract methods
9	Total number of methods containing try and catch
10	Total number of methods containing debug info
11	Number of methods in common code filter
12	Total number of static fields
13	Total number of instanced fields
14	Total size of instructions expressed in bytes

3.2.1 Features extracted.

For practical reasons, we classify our proposed features into five categories: Directly extracted features, features inferred from the strings section, features produced by opcodes, features composed from the .dex file and features not related with the .dex file. In the following paragraphs we explain them in detail and align these features with previously suggested features from other works.

Directly extracted features. These features are calculated or extracted while analyzing a .dex file. As an example, we have the number of classes, number of fields, count of instructions, etc. All these features are dependent

on the author's style of programming. These absolute values could vary from app to app because of size and functionality of the app. However, programming style about using less or more of these values remains similar as we observed during our experiments. Table 3.3 shows these features.

Table 3.4: Inferred features from string section.

Feature	Description
Reflection	Reflection property enable to extract class information at run time, like methods name. This property is used in legitimate apps for backward compatibility, or in malicious apps to hide a suspicious method's call.
Crypto	Java provide its own cryptography libraries. These libraries are not used by default, an author should decide to use it in her app.
Strings at runtime	One way to hide suspicious strings is to create them at runtime. To be able to create strings at runtime, author should use StringBuilder library, which is part of Android Framework.
Dynamic code	Legitimate or malicious apps can load more code from a .dex file, a .jar file or even another .apk file inside of the app.
Native code	Native code runs faster because it should not be interpreted by dalvik VM. It is also used to hide functions in an app, and make analysis more dificult.
Known algorithms	Some apps may use well known algorithms, quicksort for example. These algorithms are author's choice, or selected library. We manually created a list of known algorithms to look for. See Apendix for the full list.
Known data structures	Some apps may use well known data structures, as such as tree, vector, map, hash and their combinations. These data structures are author's choice, or included in selected library. We manually created a list of known data structures to look for. See Apendix for the full list.
Unicode strings	Usage of unicode symbols or strings in text strings of an app is an author's preference, usage of unicode or non-ascii values in methods' name are result of a tool used to obfuscate apps.

Features inferred from the strings section. This group includes metrics and information that is present only in the string section of a .dex file. We are not using directly the extracted strings as features; however, we are extracting useful information from them. These proposed features are influenced by the author's preferences or practices, which includes techniques, algorithms, data structures and toolchain used. The strings section in a .dex file contains different types of strings, such as types, class names, method names, and text used in an app; some of them are created automatically by the Android development toolkit when compiling an app, i.e., text strings from Android framework, types of fields, renamed method name after obfuscation. However, other strings depend completely on the developer; i.e., activities' name, text string hard-coded in the app, class and package names. Therefore, instead of analyzing all those strings as a sequence of characters, we use information from them to infer new features. We use information that include the existence of specific strings, library names or combinations of them. This group of features and their individual descriptions are shown in Table 3.4. It should be also noted that these features have different formats and values, i.e., numeric, boolean and text formats.

One of the recommended features from Spafford and Webber in binary files is to identify which data structures and algorithms are been used. As Android apps are created in the Java programming language, data structures used directly on the code are arrays, the rest of data structures should be instantiated from Java libraries. To detect the use of different data structures than

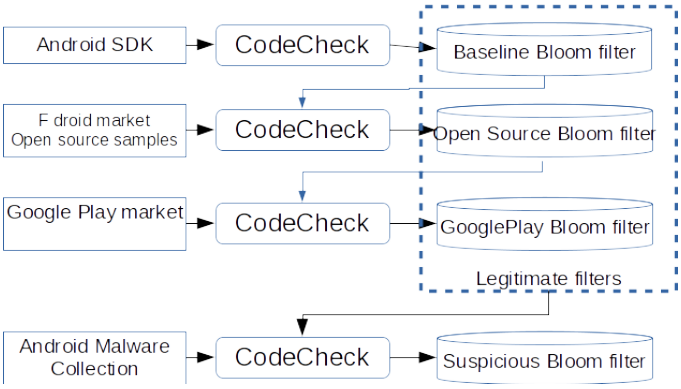

Figure 3.3: The process of building Bloom filters

arrays in an Android app, we analyze its strings to find if such structures had been instantiated. As the Java programming language offers libraries for well known algorithms, we create a list of these algorithms and look for any references to identify if the app is using any of those algorithms from libraries.

Features composed by opcodes. Main binary or executable code in an app is located in the .dex file, this file contains several sections, one of those sections contains method's information. This information includes the actual code to be executed, and it is composed of two parts: operators (opcodes) and operands (parameters). Extracting operators from a .dex file implies parsing a file, locating all method sections and understand the bytecode to keep only operators. This process results in a long sequences of bytes.

To expedite the analysis and improve scalability of our process, we develop an approach to filter known and widely used code [53]. This technique is

Table 3.5: Features composed by opcodes.

Feature	Description
Opcodes	Android opcodes are one byte long. From 256 possibilities of a byte long value, only 12 values are not used as valid opcode. The rest generate a sequence of opcodes to be processed in further steps.
Data Structures	Array data structures are the only one used and represented directly in opcodes. The rest of data structures used should be defined as objects or from libraries. We extract only 16 opcodes related to this data structure, also we extract the size of an array data structure, adding more diversity to this feature. Certain adversarial styles abuse of array data structures to hide code, perform transformation or as white box encryption mechanisms.
Number of arrays used	While extracting data structures related opcodes, we also extract the number of arrays used in an app. This is a numerical feature that shows some specific style of an author, related in preferences on array data structure usage.
Opcode groups by functionality	We also propose to map all the valid opcodes to 19 respective functionality groups. These groups are able to represent an overall view of code in a compact form, without loosing all semantic meaning. A sequence of functionality groups is produced as feature.

used to reduce the amount of data needed to be analysed from each .dex file, discarding known and already cataloged methods from the ADT framework and well known libraries. Our "common code analysis technique" is based on the knowledge that Android apps are self contained and they include Android framework support, third-party libraries, and also other public available code into their .dex files. In some cases, over 80% of the total existent code in an Android app is reused code from those sources. To define and identify which methods are common, we defined a procedure to extract opcodes from each

method, then we hash the list of opcodes for a method creating a unique identifier for it. As an app can contain hundreds or thousands of methods, we obtain a list of identifiers for each method contained in an app. After analyzing a specific group of apps, we can decide a threshold (usually 10%) to identified methods that are present between those apps. As an example, a *init* method which calls the init of a parent class is very common among apps and if it appears in more than 10% of the total apps it is identified as known basic common code. When analyzing an app, we proceed to create this unique identifier for each method and search for it in our containers to detect it as "common code" or "unique code". If it is identified as unique code, further process or analysis is conducted.

After experimentally finding which methods are common, we store and catalog identified methods into one or several bloom filters. An overview of the process to create those filters is showed in Figure 3.3.

We employ four different groups of apps with certain specifications. First we create a baseline filter from basic apps created from Android sample code, as classic "Hello world". All methods outside of the main class in each app were considered methods from Android Framework. A second group containing apps from github and f-droid market (an open source market) were filtered by the baseline filter, and create a new category of common code from this group. This new group represents open source code, this code does not include advertisement libraries nor analytic or other commercial libraries. A third group of apps obtained from GooglePlay market with all suspicious

apps removed, was used after having been filtered by the two previous ones. This filter represents legitimate, non malicious code used in the real world, it includes third party and commercial libraries, it also includes advertisement libraries. Finally a forth group of malicious apps was used to find common methods in Android malware. More detail about these filters in Section 4.2

We employ our program parsers to extract all the operators in the form of opcodes as they appear in a .dex file. Each parser perform "common code" analysis before the extraction of resulting opcodes, then these opcodes are used in different ways to generate our proposed features produced by opcodes. Table 3.5 shows and explains each of the four features in this category.
Analyzing opcodes resembles analysis of source code written by an author. Compiler, optimizer and other tools transform source code to binary code, keeping only part of the original author's style, but enough to help us to differentiate styles between developers. These differences between developers are the result of original source code and tools employed by authors to create their apps.

Features composed from a .dex file. Some features are composed from observations on .dex files, like existence or absence of some sections, and how popular or common is its structure. We create a signature based method from the observation of these elements, their existence and their position in a .dex file. While the existence of annotations, debug code and try-catch

Table 3.6: Features composed from a .dex file.

Feature	Description
Signature Group	Nominal values for: basic, normal, suspicious, dexguard or unknown.
Signature	The actual string generate with structural signature technique.
Existence of debug information	Debug information section is useful when testing apps, and should be removed from final release mode. However, not all the developers remove this section, therefore the existence of this section is an indicator.
Have gaps in .dex file	It is possible and acceptable to have empty spaces or gaps between sections, optimizers and other tools get ride of those to create a compact .dex file.

sections are already covered in previous proposed directly extracted features, existence of gaps (blank spaces) between sections is not covered yet. These gaps are formed as a result of using a tool or a particular set of configuration parameters, i.e., compiling an app in debug mode will result with big gaps between sections, while compiling the same app with optimization parameters will remove those gaps.

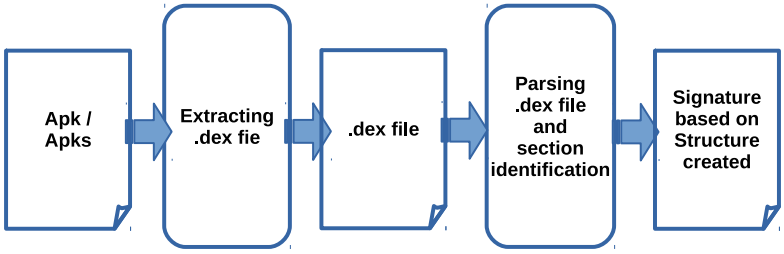

Figure 3.4: StructSignature technique description.

During the course of this work, we develop a unique technique named **Structural signature** [57] to extract a set of features composed from a .dex file.

Each .dex file is a binary container, which includes different sections with references or data related to an app. Data structures and references presented in [9] are very clear about the first sections, however the *data section* contains actual values and information pointed by other sections, i.e. encoded strings pointed by string reference section, or bytecode pointed by a method reference. Information from the same type is group together, as an example, all string values are stored together in some part of a .dex file. The order of these groups or sections is a result of the development environment and tools used to produce the .apk file. As an example of the source code, it is possible that an app does not define interfaces, so a .dex file could contain or not a group for interface definitions, however all the apps should contain string values section, but the tools used to produce a final .apk file impact the relative location of this section, e.g., if it appears before or after the code section. Another situation is the presence or not of a debug information section in an app, because this one is usually removed when using some optimization tools. One more indicator is the presence of empty spaces or gaps between sections, this is acceptable by the specification and it is removed when using any optimizer tool. To produce this signature, first we parse a .dex file, then we identify all sections by following information pointers, finally we assign a letter to each section. For some files with different content and size we can find same signature by analyzing only the .dex file layout structure. Figure 3.4 shows this process. Through our experiments we identify 5 groups of unique structural signatures: *basic, normal, suspicious,*

dexguard and unknown. Basic and normal signatures are created with ADT, with different parameters as debug mode or release mode with proguard included. Suspicious signatures were produced by different tools that employ smali to produce a new .dex file. Dexguard was the only tool that clearly produces a unique fingerprint, that is why we have a group for only this tool. Any other signatured is identified as unknown, in our empirical evaluation only 3% of .apk files belong to this group.

This classification of signatures is related to tools or parameters used to create an app, which are influenced by the author preferences. Basic and normal signatures are not very helpful for malware analysis, as they only show that a developer used normal tools. However, the rest of the signatures point to either suspicious tools or suspicious behavior, which impacts the creation of an author's profile. Table 3.6 shows features composed from a .dex file.

Features non related to a .dex file. Outside .dex file we can find other features influenced by author's preferences, tool's configuration or author's style. Permissions had been used to study Android security for a long time now, from authors perspective, we only focus on number of total permissions, and extra new permissions defined by an author. Although an app could request all permissions due to miss-configuration or lack of knowledge on developer side, new permissions should be designed and specified in app code.

Entry points as "Main Activity", "Services" and "Receivers" are decided by

Table 3.7: Features non related to a .dex file.

Feature	Description
Package name	String that identifies package name of the app, extracted from AndroidManifest.xml file.
Main Activity name	String that identifies main activity of the app, extracted from AndroidManifest.xml file.
Length of Main Activity name	Number of characters included in this string.
Entry points names	A list of strings representing entry points names.
How many entry points exists	Numerical value of entry points.
Basic permissions	A boolean vector that keeps information about basic and well known permissions requested by an app.
New permissions	A list of strings with the names of the unknown permissions requested by an app.
Is Owner the same as issuer?	Boolean value that indicates same informatino in both fields of certificate.
Validity time in seconds	Difference between start and end dates for a certificate to be valid.
Owner information	Strings extracted from owner field of a certificate.

a programmer based on functionality of an app. Also names of these activities should not be obfuscated, furthermore they keep original information from which we can extract strings representing them, and other metrics, e.g. length of those names.

A certificate also contains information influenced by the author preferences, e.g. validity period or the way it was filled. While someone could argue that a certificate actually contains information about an author or the company behind an app, it is easy to build new certificates for legitimate or malicious purposes. It is a known practice for malicious authors to create new

certificates each time they produce a new version of a malicious app. Since most of these steps are performed automatically, these selected features are consistent. Table 3.7 shows a list of features not related to a .dex file.

3.3 Feature representation.

Feature representation is the next step after feature extraction. As the extracted features need to be represented in a suitable form for a machine learning analysis. Extracted features can be in boolean, numerical or nominal form, or they can even be a sequence of strings or characters. While boolean and numerical features are easily used, nominal and string features could deliver better results as they offer more information. However, those types of features should be represented in a suitable form. Opcodes are an example of string sequence features as they are extracted from a binary or executable file. It could be hard to use them directly in a machine learning analysis, therefore they will need to be transformed into another finite and consistent form, e.g. n-grams. Different representations of the same feature could lead to compose new features, i.e., top 10 opcodes used and top 10 bi-gram opcodes used. A different example is time related features, instead of using time and date directly from certificates, we represent those values as a difference between them and obtain a validity time period.

Table 3.8: Features type and representations.

Feature	**Type**	**Representation**
Features that are extracted directly		
Number or classes	N	Direct
Number of classes containing interfaces	N	Direct
Number of classes containing information about source file	N	Direct
Number of classes containing annotations	N	Direct
Number of classes containing data	N	Direct
Number of direct methods	N	Direct
Number of virtual methods	N	Direct
Number of abstract methods	N	Direct
Number of methods containing try and catch	N	Direct
Number of methods containing debug info	N	Direct
Number of methods in common code filter	N	Direct
Number of static fields	N	Direct
Number of instanced fields	N	Direct
Total size of instructions expressed in bytes	N	Direct
Features inferred from strings section		
Reflection	B	Direct
Cryptography	N	Direct
Strings at runtime	N	Direct
Dynamic code	N	Direct
Native code	B	Direct
Known algorithms	St	Count vectorizer
Known data structures	St	Count vectorizer
Unicode strings	N	Direct
Features composed by opcodes		
Opcodes	St	N-grams with hashing
Data Structures	St	N-grams with hashing
Number of arrays used	N	Direct
Functionalities	St	N-grams with hashing
Features composed from .dex file		
Signature Group	No	Label encoder
Signature	St	N-grams with hashing
Existence of debug information	B	Direct

Continued on next page

Table3.8 –continued from previous page		
Feature	**Type**	**Representation**
Have gaps in .dex file	B	Direct
Features non related to a .dex file		
Package name	St	N-grams with hashing
Main Activity name	St	N-grams with hashing
Length of Main Activity name	N	Direct
Entry points names	St	N-grams with hashing
How many entry points	N	Direct
Basic permissions	B	Boolean vector used direct
New permissions	St	N-grams with hashing
Is Owner equals to issuer?	B	Direct
Validity time in seconds	N	Direct
Owner information	St	N-grams with hashing
Key values for tye of feature: B : Boolean feature N : Numeric feature No : Nominal feature St : String text feature		

The final step of feature representation is to create a single feature vector for each Android app to be analyzed, containing all the features extracted in the previous step. **Boolean and numerical features** are incorporated directly into a feature vector, while the rest are transformed. Since the focus of our approach is to create profiles capable of covering new apps, our feature representation has to be independent of the dataset. Normalization or text transformations such as IDT-TF are not suitable because they depend on the current dataset. For **nominal features**, we define a fixed set of possible values that encoded them to add to a final feature vector using *label*

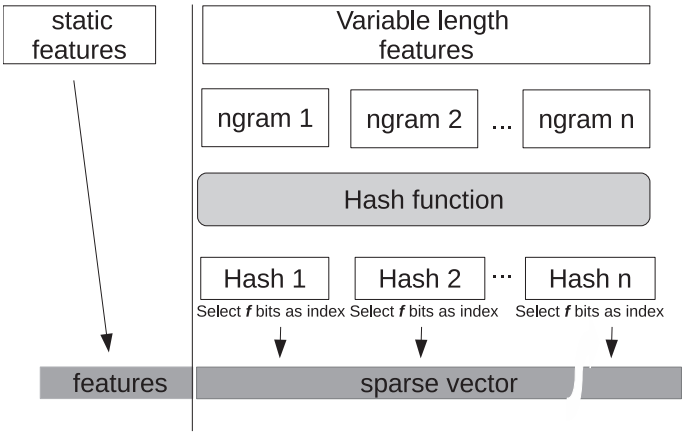

Figure 3.5: Creation of feature vector.

encoder. For **text string features**, we employ a variation of the bag-of-words technique for some features called *Count vectorizer*, and *n-gram with a hashing technique* for others. Table 3.8 shows a complete list of features, types and their representations to comprise the final feature vector.

As an example of text string feature representation, lets consider a sequence of opcodes as a feature. After extracting this sequence from a variety of Android apps, it is very probable that the number of elements in each one is different. Using the bag-of-words technique or only counting the frequency of these opcodes gives us a 234 feature vector, one value for each possible opcode. To preserve some semantic meaning from opcodes we can employ n-grams. Using 2-grams, the feature vector then contains 57,600 possible values. Using 3-grams, the feature vector increases to 13,824,000 values. From a 4-gram representation, the feature vector contains 3,317,760,000 possible values. To avoid this very high dimensional vector, we employ a hashing

technique to reduce the number of values in the feature vector. We define a size for the feature vector x, usually a power of two will work better, then the resulting n-gram token is hashed and the last x bits will indicate its index position in a sparse feature vector. More detail on this technique to embed strings is found in a study by Rieck et al. [109].

Figure 3.5 show a simplified version of the process to construct our final feature vector ready to be used in our proposed framework.

3.4 Attribution and Discovery approach

Since the goal of the proposed approach is to systematically attribute Android apps to corresponding author's profiles beyond the already known ones, we developed an incremental method to discover new author's profiles. We attribute apps based on stylistic discriminators when possible to our set of candidates or known author's profiles. If a new app could not be attributed, but is stylistically similar to other also not attributed apps, they might potentially represent a new author and form a new author's profile.

The proposed framework incorporates two components: a *Reference profiles* focusing on creating initial profiles for known authors, and an *Emerging profiles*, responsible for the ongoing analysis of Android binaries, their attribution to existing reference profiles and the generation of new profiles for stylistically similar binaries. An overview of this approach is given in Figure 3.6.

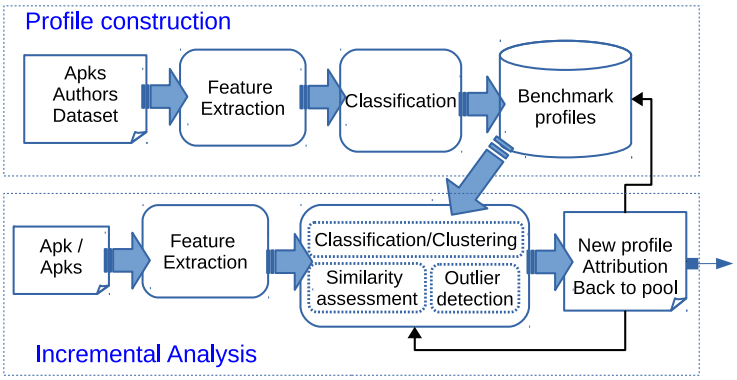

Figure 3.6: Attribution and Discovery approach.

Since we consider a large feature vector containing stylistic features (over 25,000), we are affected by the curse of dimensionality, as a result in our initial experiments clustering algorithms do not behave correctly. We adapt a classification algorithm to perform novelty detection and to discover and create new classes based on similarity with other profiles.

Reference Profiles. The first component is responsible for generating a set of author's profiles known as *reference profiles*, which will be used for further analysis. This step can be expressed as a supervised authorship attribution problem; that is given a set of Android authors $A = \{a_1; a_2; a_3; \ldots; a_n\}$ and their respective apps $A_1 = \{app_1; app_2; app_3; \ldots; app_m\}$ to generate a model that can be used to attribute Android binaries to a set of these authors. The aim is, given an apk file *app*, to determine who among these authors wrote it. We use machine learning boosting techniques to build this model. These techniques have the capability to offer a probability value as a

result. For example, element a is 53% like class Z instead to only classified element a as class Z; this gives us a better idea how similar a is to the rest of the items in class Z. Our approach uses predicted probability to discover and create new author profiles.

Although we experiment with different algorithms (See Section 4.4), Random Forest was chosen. The Random Forest classification algorithm was proposed as a bagging technique with an additional layer of randomness. When building standard trees, each node is split using the best split among all variables. In a Random Forest each node is split using best among a subset of variables randomly chosen at this node. Although this strategy might seem counter intuitive, it is quite robust and turns out to perform very well even in the presence of over fitting [17]. The primary focus of the Reference Profiles model is to attribute Android binaries to a given fixed set of authors.

Emerging Profiles Once the model outlining reference profiles is generated, the next step is to incrementally build new profiles. We designated a mini-batch algorithm that attributes new binaries to the existing profiles. It also helps discover new candidate profiles for Android binaries that were not attributed yet to existing authors. This process is outlined in Algorithm 1. The algorithm takes as input apps to be processed (*apps*), thresholds (*benchmarkThreshold*, *incrementalThreshold*) and *buckets* to group similar apps. It offers as output apps attributed (*appsAttributed*) and apps to create new profiles ($appsNewProfiles$).

Algorithm 1 Minibatch Cluster and Classification algorithm

```
1: function Incremental profile discovery
2:   input : apps
3:   input : incrementalThreshold
4:   input : benchmarkThreshold
5:   input : buckets
6:   output : appsAttributed, appsNewProfiles
7:   for each app in apps do
8:     Calculate ProbabilityScore
9:     if ProbabilityScore > benchmarkThreshold then
10:      Add app to appsAttributed
11:    else
12:      Grouping app into buckets
13:    end if
14:  end for
15:  for each bucket in buckets do
16:    if number of apps in bucket > incrementalThreshold then
17:      Remove outliers from bucket
18:      Create NewProfile from bucket
19:      add NewProfile to appsNewProfile
20:    end if
21:  end for
       return appsAttributed, appsNewProfiles
22: end function
```

The assignment of new apps to a *bucket* is based on the probability score of an *app* belonging to any of the existing profiles, or similarity between incoming apps, if a new profile needs to be formed. Following benchmark profile construction step, we employ the Random Forest classification to calculate a probability score of a given app being a part of each of the benchmark profiles. However, any machine learning method that gives a probability can be employed in this step.

If a probability exceeds a defined $benchmarkThreshold$ (we experimented with several values, however we set up at 70%), we attribute this instance to a predicted author profile (line 9), otherwise this probability is used to group apps into a new possible candidate profile (line 12). In other words, if app is scored to be a part of profile $A1$ with 98%, it is automatically assigned to that author. However, if the probability score is 20%, then it is grouped with other apps that are also given 20% probability of being a part of profile $A1$. The intuition behind this is simple: even though these apps are not close enough to profile $A1$, they share common attributes and can potentially form a new candidate profile. How the apps are grouped is guided by bucket size, e.g., with a bucket size of 10, for any given profile apps with a score 0-10%, 10-20%, 20-30%, etc. will be grouped together.

The next step is to assess how similar these unassigned apps are within their respective groups and filter out potential outliers (line 17). The similarity between grouped apps is calculated based on Cosine Similarity ($K(X, Y)$) which computes the normalized dot product of X and Y as showed in Equa-

tion 3.1.

$$K(X,Y) = \frac{\langle X,Y\rangle}{(\|X\| * \|Y\|)} \tag{3.1}$$

To define outliers, first we found maximum distance between instances, calculate average, standard deviation, and quartiles $(q1, q2, q3)$. Then we decide on an outlier threshold between an intuitive value define in Equation 3.2 and an upper outlier threshold based on the standard box plot rule, defined in Equation 3.3.

$$outlierThreshold = \frac{maximun_distance + average}{2} \tag{3.2}$$

$$outlierboxplot = q3 + iqd * 3 \tag{3.3}$$

In Equation 3.3 $q1, q3$ represent lower and upper quartile, and $iqd = q3 - q1$ represent interquartile distance.

Finally, an outlier is flagged if more than half of similarity distances are over this outlier threshold. We experiment with different outlier thresholds such as maximum or minimum of the previous values, and with no outlier detection at all.

If the number of elements in the resulting group of apps is over this incremental threshold (8 apps was defined as threshold 4), it forms a new candidate profile(line 18). It is important to note that for this probability-based grouping it is necessary to avoid expensive pair-wise comparison among all

unassigned apps. Apps not attributed in this round, are assessed in the next one, this time including new candidate profiles discovered in the last round. If no new candidate profile is discovered, the system is paused until it receives a new stream of apps to test and discover, incorporating these new ones to the previous apps not attributed yet.

At that time, our model is retrained to incorporate new candidate profiles discovered in the previous step. Note that as opposed to benchmark profiles that include validated information about their authors, these new profiles are limited to a label that is created automatically from originating profiles, probability and number of apps in a newly created group. Further analysis for features extracted from those apps forming this new profile could be manually conducted.

3.5 Prototype implementation

In this section we explain details about our proposed approach implementation, architecture and challenges faced during developing. This framework prototype includes 7 command line user interfaces, 13 modules, above 160 functions and nearly 4800 lines of python code.

3.5.1 Architectural view

Our proposed framework architecture is modular. The code base is split into several independent components that can be used independently. This

ensures separation of components that communicate among themselves, it also permits inclusion of more components or reuse of individual components with other projects. Our components, named modules, can be roughly split in five main categories.

1. **Filter category** includes apk validation and Android packer detection modules.

2. **Feature extraction category** includes modules related to processing valid .apk and .dex files, and feature extraction. All currently employed features are handled in modules within this category. Future features that would be potentially introduced would also be included here.

3. **Feature representation category** includes modules that convert featrues into suitable format.

4. **Attribution engine category** includes modules to create reference profiles and modules to attribute and discover new profiles.

5. **Miscellaneous category** include communication modules between client and server, storage and cache modules.

Each of this category holds several modules as shown in Figure 3.7. Let us consider these modules grouped in packages, where each package accomplishes a specific step in our framework. The modules communicate among themselves passing data and messages using local or remote methods.

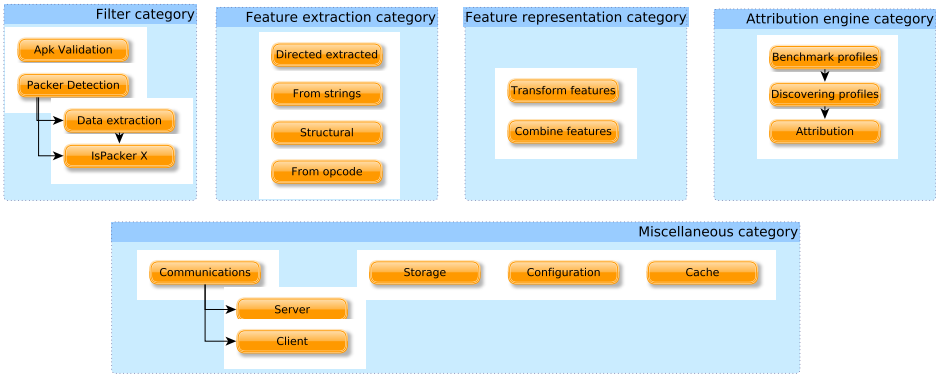

Figure 3.7: Five categories holding different modules of our system.

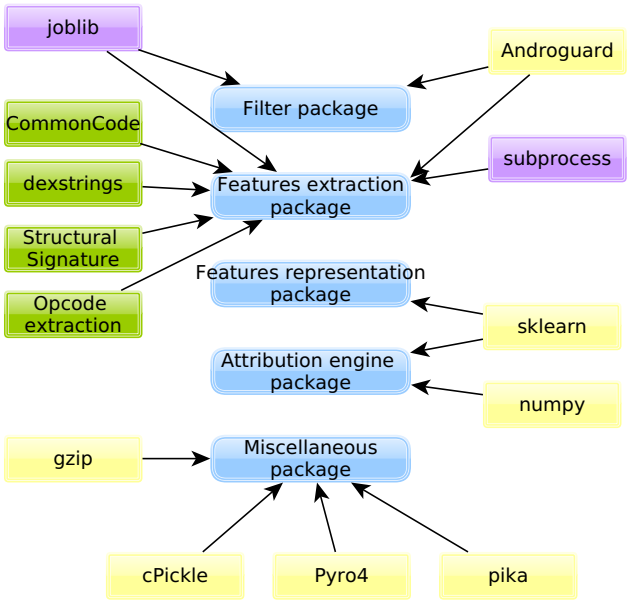

Figure 3.8: External modules related each package of our system.

The overall framework was developed with the Python language, using specialized libraries such as sklearn, numpy and androguard to accomplish specific tasks through the process. These specialized libraries, only interact with certain packages in of our framework, offering their functions through them. Graphical view of the relations between modules and libraries are shown in Figure 3.8. Green libraries represent software components that were developed during the course of this research.

Filter package modules employ slightly *Androguard* [36] framework in the validation of apk files. This package also employs *joblib* library to perform parallel processing of several apps.

Feature extraction modules employ also *Androguard* to extract information about permissions and activities. They also employ *joblib* and *subprocess* libraries. In this package, several external projects were integrated as libraries to participate during feature extraction steps. These software components were developed during the course of this work research, and released as open source software tools:

1. CommonCode [53] (source code available at [55]).

2. StringAnalysis. (source code available at [56]), which employs dexstrings (source code available at [60]).

3. StrucSignatures (source code available at [57]), which employs Android-Soo [61] (source code available at [58]) and DroidColors [73] (source code available at [59]).

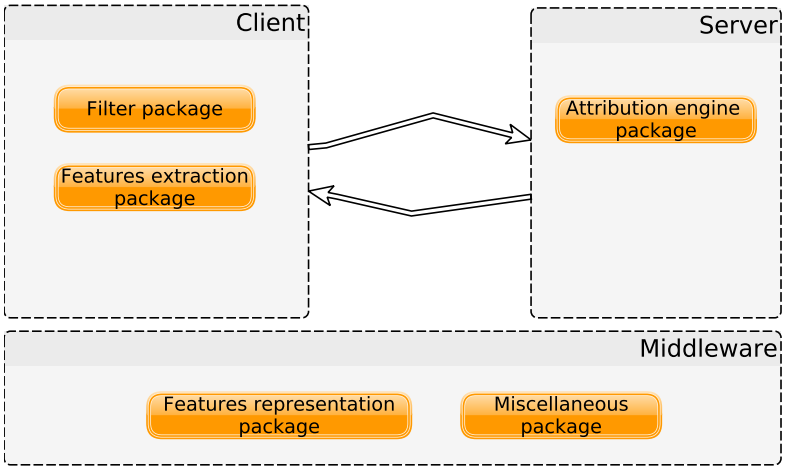

Figure 3.9: Client-Server architectural view of the proposed framework.

Also several .dex file parsers developed in C programming language wrapped with Python interfaces where employed to extract features from bytecode.

Feature representation modules are closely related with feature extraction modules. For each extracted feature, a convenient representation should be prepared.

Attribution engine modules employs *scikit-learn* [100] framework, which is the standard framework for machine learning tasks in Python. These modules also employ *numpy* library that offers high performance for numerical computations.

Miscellaneous modules employ *Pyro4* and *pyka* libraries for managing client/server communications. For creating cache files and storing information these modules employ *gzip* library for data compression and *cPickle* library for data serialization.

Our proposed framework is designed with a Client/Server architecture in

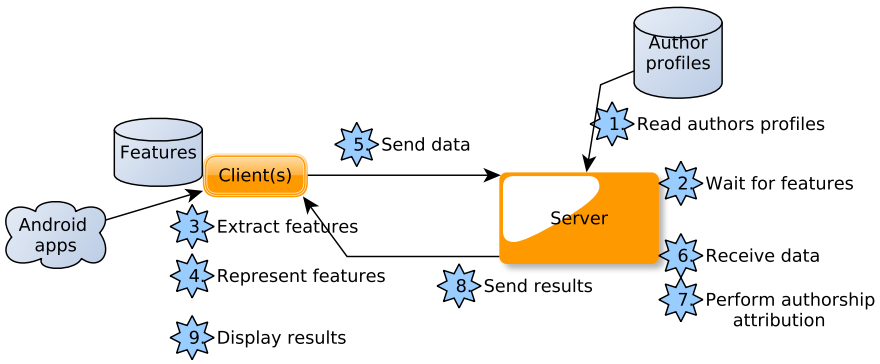

Figure 3.10: Overview of the attribution process.

mind to offer scalability and flexibility. It can be used as typical client-server software or as a distributed system to spread out the working load of first stages between several computers. Figure 3.9 shows client-server architectural view of our proposed framework, it includes a middleware section which provides communications, caching and storage capabilities to the system. Its also possible to use feature representation on the client or server side, depending on the load of our system. Client side performs filter step and feature extraction. Server side main job is to perform attribution and discovery, and storing information about profiles.

Our framework implements three basic different systems: 1) Creation and evaluation of *reference profiles* that are used as initial input to the rest of the systems. 2) Discovery and attribution process, which use reference profiles as initial input and increase the number of known profiles while discovering new ones. It includes options to evaluate results also. 3) Attribution only process, which can use as initial input any set of author profiles created previously.

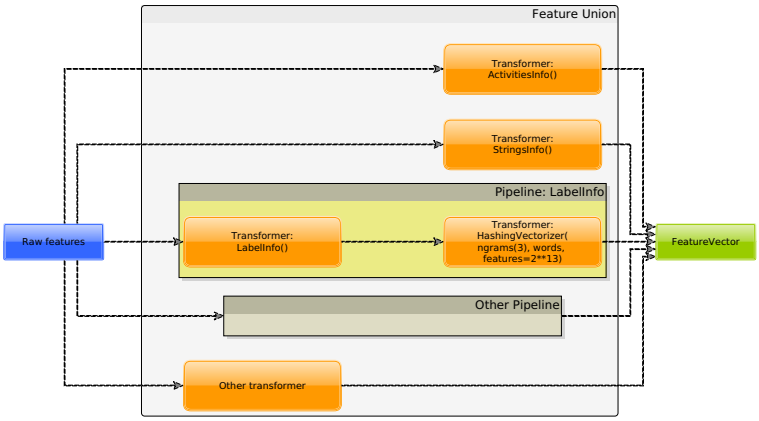

Figure 3.11: Overview of the feature union process.

This system does not perform discovery, only authorship attribution. The process for this system is shown in Figure 3.10.

A technical challenge was how to put together all the features extracted in a suitable format, for this task after creating representations of features, we employ *FeatureUnion* and *Pipe* functions from scikit-learn framework to process raw features and produce a single feature vector for each app. Figure 3.11 shows a reduced version of how this process works. **FeatureUnion** is used to concatenate all raw features after being processed. The **transformer_list** structure contains methods to process raw features and create adequate representations; i.e., *activitiesInfo()* method reads the raw feature file produced, and it extracts information related to activities as the total number of activities, the number of services, etc, then produce a numerical vector which is used in the final feature vector. One method should be developed for each transformation. Some transformations are complex or with a

```
XFeatures =  FeatureUnion(

transformer_list=[
('ActivitiesInfo', ActivitiesInfo()),
('StringsInfo', StringsInfo()),
('ArrayClassInfo', ArrayClassInfo()),
('ArrayLenInfo', ArrayLenInfo()),
('OpcodesCommonInfo', OpcodesCommonInfo()),
('OpcodesPercentajeInfo', OpcodesPercentajeInfo()),
('PermissionsInfo', PermissionsInfo()),
('CertInfo', CertInfo()),
('SignaturesInfo', SignaturesInfo()),
('ArrayInfo',  Pipeline( [
('selector', ArrayInfo()),
('CountVectorizer', CountVectorizer(ngram_range=(1,1),
analyzer='word',
vocabulary = vocArrays),
),
]),
),
('LabelInfo', Pipeline( [
('selector', LabelInfo()),
('CountVectorizer', HashingVectorizer(ngram_range=(1,3),
analyzer='word',
n_features=2**13,#,
non_negative=True #,
),
),
]),
),
('StringsStructuresInfo', Pipeline( [
('selector', StringsStructuresInfo()),
('CountVectorizer', CountVectorizer(ngram_range=(1,1),
analyzer='word',
vocabulary = vocStructures),
),
]),
),
('StringsAlgorithmsInfo', Pipeline( [
('selector', StringsAlgorithmsInfo()),
('CountVectorizer', CountVectorizer(ngram_range=(1,1),
analyzer='word',
vocabulary = vocAlgorithms),
),
]),
),
('ActivitiesMainInfo', Pipeline( [
('selector', ActivitiesMainInfo()),
('HashTrick', HashingVectorizer(ngram_range=(1,3),
analyzer='char',
norm=None,
n_features=1024,
non_negative=True),
),
]),
),
('ActivitiesPackagesInfo', Pipeline( [
('selector', ActivitiesPackagesInfo()),
('HashTrick', HashingVectorizer(ngram_range=(1,3),
analyzer='char',
norm=None,
n_features=1024,
non_negative=True),
),
]),
),
('PermissionsNewInfo', Pipeline( [
('selector', PermissionsNewInfo()),
('HashTrick', HashingVectorizer(ngram_range=(1,3),
analyzer='char',
norm=None,
n_features=1024,
non_negative=True),
),
]),
),
('SignaturesSigInfo', Pipeline( [
        ('selector', SignaturesSigInfo()),
        ('HashTrick', HashingVectorizer(ngram_range=(1,2),
                                        analyzer='char',
                                        norm=None,
                                        n_features=512,
                                        non_negative=True),
                                        ),
    ]),
),
('CertOwnerInfo', Pipeline( [
        ('selector', CertOwnerInfo()),
        ('HashTrick', HashingVectorizer(ngram_range=(1,3),
                                        analyzer='char',
                                        norm=None,
                                        n_features=1024,
                                        non_negative=True),
                                        ),
    ]),
 ),
('OpcodesInfo', Pipeline( [
        ('selector', OpcodesInfo()),
        ('HashTrick', HashingVectorizer(ngram_range=(3,3),
                                        analyzer='word',
                                        norm=None,
                                        n_features=2**14,
                                        non_negative=True),
                                        ),
    ]),
 ),
  ],
 )
```

Figure 3.12: Python code for producing a feature vector from an Android app.

variable length results, in this case, we employ **Pipeline** to collect data and then to transform this piece of collected data; i.e., *LabelInformation* Pipeline first extract information about functional labels bytecode from raw features, then create n-grams of size 3, reducing to a fixed number of features using the hash trick. The order of the transformations is important for the produce final feature vector.

The code written for this task plays an important role when introducing new features to our framework, or when testing with a smaller set of features. Figure 3.12 shows Python code to perform the task of feature representation. Machine learning algorithms that offer prediction probabilities implemented in our framework are: Random Forest, Extremely Random Trees, eXtreme Gradient Boosting, Adaboost and SVM. We experiment with all of them when evaluating *reference profiles* to choose the one that provides the best accuracy and performance.

3.5.2 Implementation challenges

Developing a robust system is always a challenge. Here we present three of the most interesting challenges we faced during this process.

Managing unicode strings. We are analyzing apps from different sources that include non-English speaking countries. These apps include unicode strings in text files, activity names or certificate information. Our .dex file parsers written in C language can deal with unicode strings without any

trouble, but the Androguard framework and the Python interpreter presented some issues when handling unicode strings from methods or classes names from the apk when they were instanced to Python objects. These issues are caused because Python can not handle unicode characters by default in object names. It also causes effects when an apk certificate information contains non-ASCII characters in "owner" or "issuer" section because it breaks the internal information flow. This was a hidden issue affecting only the feature extraction process, and provides with wrong results. Adding unicode support to each script part of the system was not enough. To resolve this issue it was necessary to perform an undocumented and uncommon procedure of reloading the Python *sys* library after enabling unicode support to it.

Continuous work of the system. In a perfect world our system should not have disturbances while running. However, in the real world different things can happen, from a power outage, software crash, or disk failure. In early stages of our research we faced a power outage that resulted in the loss of several days' work from the server side package. To overcome this issue, we implemented a "save point" functionality similar to that of video games. In this "save point" we store results from our system after a stream of apps was analyzed and new profiles were discovered. In the case of an interruption, we have several intermediate starting points other than initial "reference profiles". However, the work of the stream of apps under analysis after a "save point" is lost. We take advantage of this solution for different

experiments with same initial point other than "reference profiles".

Repeating computational expensive work. When we were performing large scale experiments, the feature representation step consumed a significant portion of the time, because the analysis, operations and transformations it must perform. We try to optimize code and employ other optimization methods for Python, but the time did not show major improvements. We opted for an alternative solution to perform feature representation steps only as needed. If experiments where conducted on the same set of features and Android apps, we perform only one feature representation step in the client side and store output as an intermediate result. In case we are analyzing or performing authorship attribution over only one of a few Android apps, feature representation intermediate storage is not necessary.

3.5.3 Technical details of developed software components

A number of the software components developed during the course of this research and were integrated as libraries in our framework, the majority of them appeared in published works. A few supporting components however were not included in these publications and required more detailed explanations which we presente here.

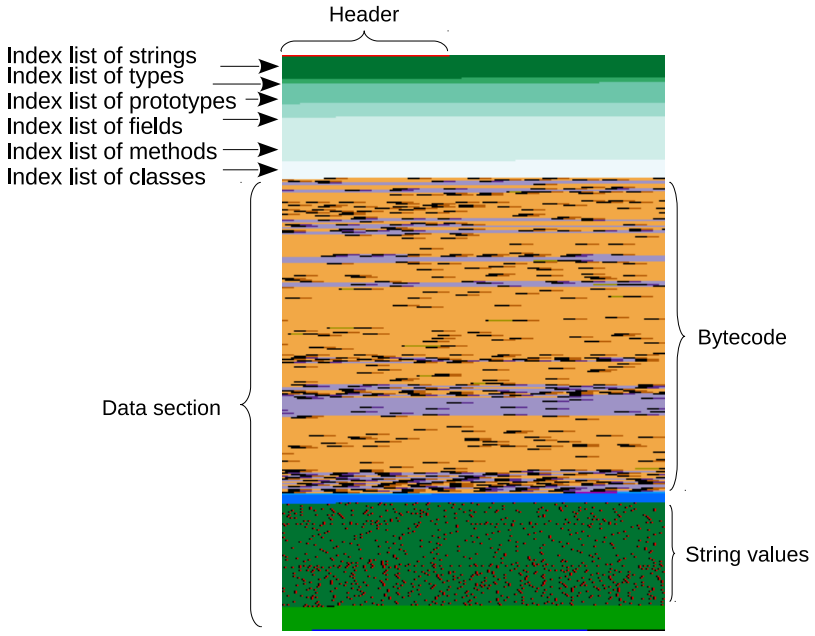

Figure 3.13: Visualization of a .dex file structure.

DexString. DexString is a special .dex file parser developed in C programming language that only focuses on the string section defined in the header. Its main goal is to extract all strings defined in a .dex file. To accomplish its goal, DexString analyzes the .dex header to find the strings offset section, then it visits each offset to read the length of the string and the actual value content. It counts how many times and with which role a string is referenced. For example, it counts if a string is used as *type definition* role by being referenced in table definition table, or if it is a *text string* used by the app and not been referenced in any other table. It also presents if a string contains unicode. The final string value is presented between .: and :. to include non-ascii characters as new line. Figure 3.14 shows an explained sample of a

```
hugo@pavilion\$ ./dexstrings \~/AndroidStudioProjects/Test1/app/classes.dex -sru
String reference | type | number of references | string offset | ascii size | unicode size | unicode | string value
(Examples of prototype)
1625 | PPP | 3 | 11bdb3 | 3 | 3 |     |.:LIF:.
1626 | PPPPPPPPPPP | 12 | 11bdb8 | 3 | 3 |     |.:LII:.
1627 | PP | 2 | 11bdbd | 4 | 4 |     |.:LIII:.
1628 | PPPPP | 5 | 11bdc3 | 5 | 5 |     |.:LIIII:.
(Types)
3227 | T | 1 | 13065b | 27 | 27 |     |.:Lcom/gsg/test1/BuildConfig;:.
3228 | T | 1 | 130678 | 28 | 28 |     |.:Lcom/gsg/test1/MainActivity;:.
(Fields)
3533 | F | 1 | 132d16 | 24 | 24 |     |.:METADATA_KEY_USER_RATING:.
3534 | F | 1 | 132d30 | 19 | 19 |     |.:METADATA_KEY_WRITER:.
3535 | F | 1 | 132d45 | 17 | 17 |     |.:METADATA_KEY_YEAR:.
(Source code references)
3879 | SSSS | 4 | 134d85 | 22 | 22 |     |.:PopupWindowCompat.java:.
3880 | S | 1 | 134d9d | 28 | 28 |     |.:PopupWindowCompatKitKat.java:.
(Calls to init method)
8015 | MMMMMM | 6 | 147ffd | 4 | 4 |     |.:init:.
(Text strings)
12941 | S | 0 | 15b3ca | 9 | 9 |     |.:yduration:.
12942 | S | 0 | 15b3d5 | 4 | 4 |     |.:yoff:.
12943 | S | 0 | 15b3db | 4 | 4 |     |.:yvel:.
```

Figure 3.14: Modified example of DexStrings output.

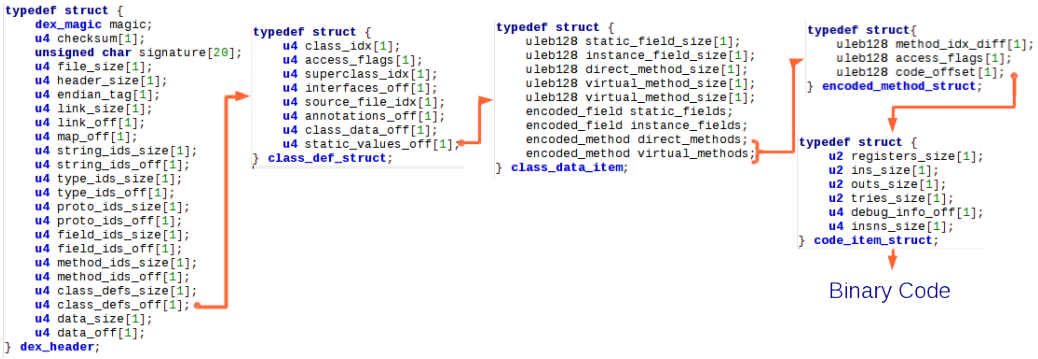

Figure 3.15: Overview of the path to parse class information to binary code.

modified DexStrings output.

Parsers for .dex file. Since there were not publicly available .dex file parser, four of them where created to extract specific feature information from .dex files.

To analyze and parse a given .dex file, we must follow the formal specification of the .dex file format [9]. A diagram of how to parse a class_item to

all its components is shown in Figure 3.15. We extract the number of classes and the offset of the class definition from the header. For each class definition we read associated values, including class identifier, parent class identifier, access flags, interfaces info, annotations offset if exists and class item info offset. From the class item info offset we read the number of fields and their information in encoded format, also the number of methods (private and virtual) and their information in encoded format, which includes identifier, access flags and the code offset for each method. The code item information includes the number of registers, input and output registers, try-catch information, debug information offset, how many instructions for this code followed by the bytecode itself. This process is repeated for all classes.

StructSignature. StructSignature was build on top of Droidcolors. This software takes a .dex file as an input and parses the header information and rest of the elements to clearly identify sections. Structures presented in [9] are very clear about the first sections, but not about the data section. This section contains actual values pointed to by other sections, like strings values referenced in string offset section. We can observe different groups of values in *Data* section; like the actual string values, class definition values, method's code, interfaces values, etc. As the same type of information is grouped together, we call those groups also sections. The order of these sections is the result of a chain-tool used to produce the .apk file, plus the actual content and structure of the app; as an example, it is possible that an app does not

contain interfaces, but all the apps should contain a string values section. From our experimental results we observed that some tools produce different layouts, as an example, it is common that string values section is located after the method's code and the class definitions sections; however, some specific tools (like dexguard) place this string values section before the method's code section. Another observation is the existence of the debug information section, this information is usually removed after using optimization tools for the .apk file.

Once we identify all sections from a .dex file, we represent them in a text string. We considered the existence and the order of these sections to create signatures. Using this signature we can group apps with similar tool-chain origin. During our research work we discovered 5 groups: basic, normal, suspicious, dexguard and unknown.

3.6 Summary

We are aware of not using any feature selection in the process, and also we are aware about the distribution or weight of the features employed. This decision was entirely conscious to avoid over-fitting and the creation of bias from original datasets, impacting future features extracted while discovering new possible profiles. We consider that using a boosting algorithm, the feature selection is carried on at the same time that classification is performed.

Our approach is completely data-driven, it heavily depends on the quality of

the first batch of samples to create reference profiles. It uses directly machine learning techniques to create *reference profiles*. After that, it employs our proposed algorithm named *Minibatch classification and clustering* to discover and create new possible author profiles. These new discovered author profiles are created in a machine learning model, understanding or explaining them directly is not an easy task, but other methods can be used for this matter, such as employing *droidkin* system to confirm relationships between Android apps belonging to a specific profile; or the extraction of yara rules from an author profiles to be integrated in other systems. Since each discovered profile is assigned with a label, a report on the apps belonging to this discovered profile can be created from information gathered during feature extraction, or when it is possible directly from markets or other Intelligence sources available. Note that these profiles represent authors from which we do not have detailed information. We discovered a profile grouping stylistic similar apps.

The development process for this prototype Framework was incremental and modular, focusing in a specific task at a time with the final goal in mind. The prototype design evolved during this work, creating new modules and incorporating new functionalities to overcome the development challenges.

It is our belief that modules from this prototype can be used alone or as components for other projects, also that feature extraction and feature representation packages could be replaced and updated to accept other type of executable files to perform authorship attribution.

Chapter 4

Experimental results

"If your experiment needs a statistician, you need a better experiment."

Ernest Rutherford

In this chapter we describe the datasets used in our experiments, and present descriptive information collected by diverse modules from our system; i.e. percentage of packed apps in diverse markets, corrupted malware files, number of suspicious signatures. We include our validation approach and finally, we aim to evaluate several aspects of our main Android authorship attribution approach:

1. Validate the accuracy of our proposed approach in attributing apps to the corresponding author's profiles.

2. Investigate the effect of reference and emerging profiles' thresholds on accuracy of attribution.

3. Assess the ability of the discovery analysis component to attribute unknown apps and generate emergent profiles.

4. Evaluate our approach capability to attribute a stream of wild data to the existing profiles.

Experimental settings parameters The experiments were performed mainly on a desktop with Intel(R) Core(TM) i7-3770 CPU and 16 GB of RAM, running Debian 8.0 with kernel 4.0.1. We also employ a server with two Intel(R) Xeon(R) CPU E5-2420 v2 with 6 cores each and 64 GB of RAM, running CentOS 6.8 with stock kernel 2.6. For testing distributed tasks we use lightly a desktop with Intel(R) Core(TM)2 Duo CPU and 32 GB of RAM, running Debian 7.8 with kernel 3.2.

4.1 Datasets

To evaluate our proposed approach and study for the automatic creation of author label profiles, we gathered a large collection of 198,181 unique Android applications from different sources that we employed for various stages of analysis. Details are shown in Table 4.1.

To validate our approach we carefully collected and manually verified apps from 43 authors. These apps were collected from eight different markets, to identify an author we use the certificate serial number with the knowledge that a developer needs a private key to sign an app. There are some cases

however, that private keys are public or leaked, i.e. keys to sign debug apps were distributed with Android Development Toolkit, or tools to automatically sign apps where distributed with a private key; these cases result in several an unrelated apps contain same certificate serial number. As a side note, in official market is not common to find the debug certificate serial numbers, but it is common to find them in third-party markets.

Table 4.1: Datasets.

Name	**Apps**	**Description**
Legitimate authors dataset	1,734	Collected from eight different markets. This dataset contains information about 43 known authors with more than 20 apps in the markets, based on the assumption that similar certificate serial numbers leads to same author (without considering public or leaked certificates)
Malicious authors	222	Collected from Koodous system [83], this dataset contains at least 10 unique malicious apps from 10 different authors. Apps where identified as malicious in Koodous and they were verified as malicious by Virustotal service.
Hacking Team apps	131	Mobile apps identified as part of campaigns ran by this Italian company. Those apps include several developer certificates and include different versions. All apps produce a different hash values.
Partial Drebin dataset	3,181	A set of malware families from Drebin dataset with more than 20 apps each.

Continued on next page

Table4.1 –continued from previous page

Name	Apps	Description
Adware dataset	211	Three families (kemoge, shedun and shuanet) related between them. Lookout company reports [1] claimed that these families are related among them and mainly contain trojanized legitimate apps.
Ransomware dataset	408	Ransomware software contains specialized behavior to lock out users or encrypt data and ask for money from the user to free the mobile or return data. This set includes seven families: fakedefender, koler, pletor, ransombo, scarepackage, slocker and svpeng.
Fdroid dataset	1,395	Open source apps without advertisement libraries.
Whatsapp evolution	469	Official releases of Whatsapp application for Android, collected from 2012 to middle of 2015.
Basic apps	100	Basic Android projects from official samples, tutorials and github. Compiled with default settings in Android Studio.
GooglePlay-2015 dataset	4,574	Apps collected from Google Play in the middle of 2015, top popular and top new apps from all categories.
Market dataset	23,656	Apps were collected from the following 8 third-party markets: 360, 3gyu, anzhi, aptoide, mobomarket, nduoa, tencent, xiaomi. They contain legitimate software, adware and malware. This set is not labeled. We crawled these apps during 2015.
GooglePlay-2016 dataset	921	Apps collected in middle 2016. Top free and popular apps.
GooglePlay-all dataset	44,915	Apps collected in January 2016. Not only top popular or new free apps were collected this time, but all apps found when searching for common terms in the market.

Continued on next page

Table4.1 –continued from previous page		
Name	**Apps**	**Description**
VirusTotal stream	116,264	Apps provided by VirusTotal [2] stream service. All apps sent to the service to be analyzed where made available to us for a few days in 2015. The group contains normal, suspicious and malicious apps. This set is not labeled.
Total	198,181	

To ensure that unique apps were used in our first analysis, only apps that produce a unique hash value for the .dex file were retained for analysis. We also discarded from this group apps that showed symptoms of being created with a packer tool, detected by our filter step. A relationship map obtained from Droidkin [62] is shown in Figure 4.1. We have well defined and clearly identifiable groups of apps by an author; we observed only a few weak relations between some groups. These weak relations show that one app from an author maintains some similarity with other apps from another author; by removing these apps we could create isolated groups per author. We decided to leave those apps to evaluate performance with real data. Since these authors were deemed to be legitimate (verified through VirusTotal service), we also searched for authors who produce malicious apps. For this purpose we employed kooduos [83] collaborative system, collecting apps containing same certificates and flagged as malicious. This produced a set of 222 apps from 10 different authors. In addition to this, we retrieved 131 malicious apps flagged as produced by Hacking team.

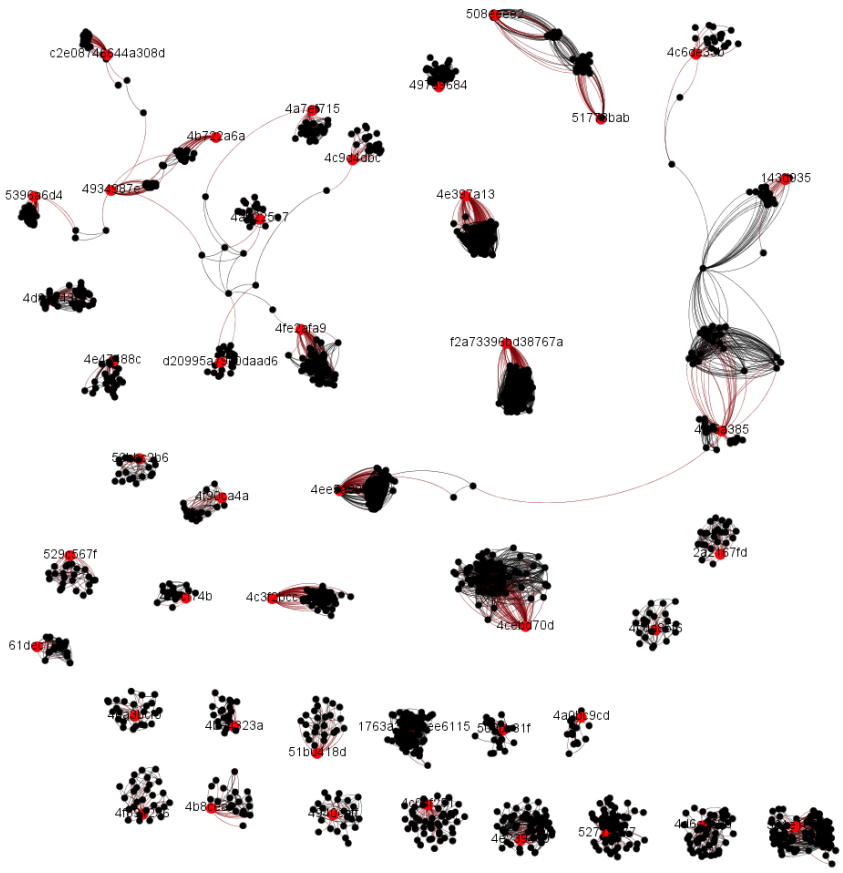

Figure 4.1: Legitimate authors dataset grouped by Droidkin. Red dots represent an author, black dots represent apps, and the lines represent relationships among them. If two red dots are connected, it represents the existance of a relationship.

To explore the benefits of our approach in practice, we also collected apps from a number of other sources. Specifically, several malware families from Drebin dataset [10]; 1,395 apps from open source market Fdroid; 10 adware and ransomware families; 4,574 apps collected from Google Play in 2015 and 921 collected in 2016; 23,656 apps from eight third-party markets.

For a large wild test, we employ 44,915 apps collected from GooglePlay in 2016. These apps were crawled searching common terms in the market and produce a large list of apps to download. Also 116,264 apps retrieved from VirusTotal stream, for a few days late in 2015. These two sets of apps are not labeled.

We also created a small set composed of basic apps from official samples. All of these 50 apps were compiled using AndroidStudio with default settings. Also 50 samples from tutorials and from github were compiled using default settings, obtaining 100 basic apps. This dataset was used to create the basic filter that contain common methods for Android apps.

The usage of these sets of apps during our experiments varies depending on the nature of the experiment. Our experiments are performed incrementally, including new datasets in each stage. However some of the sets were employed when creating the baseline tools of our framework. We identified five stages in our experiments, which were performed each after the other. In table 4.2, relations between datasets and experiment stages are showed.

Table 4.2: Relation between experiment stages and datasets.

Experiment stage	**Datasets employed**
Analysis of the author's dataset (Section 4.2)	Basic apps, F-droid, Drebin and Google Play 2015 for CommonCode. Google Play 2015, Whatsapp, F-droid, Drebin, Markets for StructSignature. Legitimate authors and malicious authors.
Reference profiles (Section 4.4)	Legitimate authors and malicious authors.
Tuning thresholds (Section 4.5)	Legitimate authors and malicious authors.
Evaluation of emergent-profiles (Section 4.6)	Legitimate authors and malicious authors for evaluation over labeled dataset. Hacking Team, Adware, Ransomware and Drebin for evaluation over malware families. F-droid, Google Play 2015, Google Play 2016 for evaluation over legitimate apps.
Experimenting with attribution only approach Section(4.7)	VirusTotal stream for authorship attribution over a large unknown stream of suspicious apps. Google Play all and markets dataset for authorship attribution in the Markets. Drebin dataset for authorship attribution over a stream of known malicious apps.

4.2 Analysis of the authors' datasets

Our authors' datasets are a crucial part of our system because they represent the original seed for the discovery and creation of new author profiles. We already mentioned how we collected those apps in the previous section, now we present in this section useful information to understand those groups of apk files better. In our proposed system the two initial steps are "filter" and "feature extraction". For the "filter step" we present the number of valid apk files detected. For "feature extraction step" we present results for a technique to reduce the total amount of code to analyze, and also we present

results related with the developer toolchain detected. These results impact in the feature extraction process and are worth analyzing them here.

Packed and damaged apks detection The first step before analyzing input apps is to filter them to discard non-valid apps. An Android app is discarded because of the lack of AndroidManifest.xml or classes.dex files, or because of any problem related to the zip file format. This step also identifies apps detected as created by a packer tool.

From our legitimate author dataset, the "filter step" discarded 154 apps (8.8%). Those apps were discarded because they were identified as created by Tencent packer tool (102 apps, 5.88%), or because they were damaged zip files or did not contain classes.dex file (51 apps, 2.94%), or because of other errors (1 app). A total of 1580 apps (91.12%) continued to the next step.

For the malicious author dataset, a total of 224 apps (84.21%) continued to the next step. From the discarded apps, 37 (13.91%) were identified as packed by Tencent packer and five (1.88%) were damaged zip files.

A summary of results is showed in Table 4.3.

Reused code detection Android apps are self-contained commonly, including libraries and framework code in their .dex file. The aim of detecting this common and reused code between apps is to lighten subsequent analysis by omitting methods when we already know their origin. To achieve this goal we employ our tool **CommonCode** [55] to create four filters for our proposed framework. Table 4.4 shows filter name, input dataset, conditions

Table 4.3: Experimental results from filter step for legitimate and malicious authors.

Total Apps	Pass	Fail	Packed	Damaged	Other
For legitimate 43 authors.					
1734	1580 (91.12%)	154 (8.88%)	102 (5.88%)	51 (2.94%)	1(0.06%)
For malicious 10 authors.					
266	224 (84.21%)	42 (15.79%)	37 (13.91%)	5(1.88%)	0

to create it, and the objective of this filter. We can observe that this process reduced the code to be analyzed to only 12.07% in the legitimate author dataset, and to 22.66% in the malicious author dataset.

The results of applying these filters are showed on Table 4.5. Basic filter and opensource filter discarded consistently around 60% and 10% in both groups. The legitimate filter was almost double with 18% versus 9% respectively. Suspicious filter is lower than 1% in both cases. Used in this way, this technique is not able to differentiate between legitimate and malicious apps or authors, but helps to reduce the amount of code to be analyzed in further steps.

Structural signatures analysis Structural signatures offer an inside view of the tools used to create Android apps. We employ five groups to classify apps: basic, normal, suspicious, dexguard and unknown. Signatures from unknown group offer a degree of uniqueness for those apps. Structural signatures were produced by our tool **StructSignature** [57]. The distribution

Table 4.4: Filters created with CommonCode tool.

Filter name	Input	Conditions	Objective
Basic filter	Basic apps dataset	All methods but main activity.	Remove common code related to framework and basic libraries.
Opensource filter	Fdroid market	All methods that appear in at least1% of apps from this dataset.	Remove common code found in Open Source apps.
Legitimate filter	GooglePlay-2015	All methods that appear in at least 1% of apps from this dataset.	Remove common code found in legitimate published applications.
Suspicious filter	Drebin dataset	All methods that appear in at least 1% of apps from this dataset.	Remove common code found in suspicious apps, we do not know if the code is malicious or not, it only happens to be common in this dataset.

Table 4.5: Experimental results from created filters over legitimate and malicious authors.

Number Classes	Basic filter	Opensource filter	Legitimate filter	Suspicious filter	Unique methods
For legitimate 43 authors and 1580 apps					
24,077,007 (100.00%)	14,005,881 (58.17%)	2,656,575 (11.03%)	4,482,231 (18.62%)	25,040 (0.10%)	2,907,280 (12.07%)
For malicious 10 authors and 224 apps					
3,382,278 (100.00%)	2,000,582 (59.15%)	306,643 (9.07%)	300,254 (8.88%)	8,462 (0.25%)	766,337 (22.66%)

Table 4.6: Experimental results of signature distribution among legitimate and malicious authours.

Unique Signatures	Basic	Normal	Suspicious	Dexguard	Unknown
For legitimate 43 authors and 1580 apps					
68	1296 (82.03%)	105 (6.65%)	133 (8.42%)	0	46 (2.91%)
For malicious 10 authors and 224 apps					
30	188 (83.93%)	8 (3.57%)	12 (5.36%)	0	16 (7.14%)

of the signature groups detected are showed in Table 4.6.

The high detection of "basic signature" group in both cases offer insides about authors preferences to create apps with default tools, and without using optimization nor obfuscation. Some authors used packer tools, but those apps were detected and removed in the previous step.

4.3 Evaluation metrics

We validate our approach in two phases named: reference-profiles and tuning emergent-profiles. We also evaluate a discovery approach with the same metrics for emergent-profiles. We evaluate reference-profiles phase as a multi-class classification problem, calculating *accuracy*, *precision* and *recall*. In this scenario, *accuracy* is calculated as the fraction of correct .apk files predicted divided by the total of .apk files evaluated, as given in Equation 4.1. In binary classifications (where only two classes exist) *precision* is defined as $\frac{TruePositive}{TruePositive+FalsePositive}$ and *recall* is defined as $\frac{TruePositive}{TruePositive+FalseNegative}$. How-

ever, this is a multi-class problem where these measures can be applied to each label (in this case each author) independently, and then combined across labels. We calculate *precision* as showed in Equation 4.4 and *recall* as showed in Equation 4.5. In each case we calculate measures for each label (Equations 4.2,4.3) and then add them up to obtain final results. In all equations the following notations are used: y represents the set of predicted values, $\hat{y}$ the set of true values, y_l the subset of y with label l, $|A|$ is the number of elements in set A, $|B|$ is the number of elements in set B. If $B = \emptyset$ in Equation 4.3, the result for $R(A, B)$ is 0.

$$Accuracy\left(y, \hat{y}\right) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1\left(\hat{y}_i = y_i\right) \tag{4.1}$$

$$P(A, B) = \frac{|A \cap B|}{|A|} \tag{4.2}$$

$$R(A, B) = \frac{|A \cap B|}{|B|} \tag{4.3}$$

$$Precision\left(y, \hat{y}\right) = \frac{1}{n_{Labels}} \sum_{l \in Labels} P\left(y_l, \hat{y}_l\right) \tag{4.4}$$

$$Recall\left(y, \hat{y}\right) = \frac{1}{n_{Labels}} \sum_{l \in Labels} R\left(y_l, \hat{y}_l\right) \tag{4.5}$$

The values for these three metrics range between 0 and 1, where 1 represents 100% and is thus a desired outcome.

Because traditional metrics for evaluation of classification accuracy did not reveal enough information about how the apps were attributed, we used three additional metrics to help us assess the behavior of thresholds. For emergent-profiles tuning we evaluate the percentage of how many apps were attributed in total (AA), percentage of correctly attributed apps among all considered apps (TCA), and a percentage of correctly attributed apps among those that were attributed to some profiles (CA). Value of TCA will be the main indicator, followed by CA and finally AA. We are looking to improve overall quality of attribution, i.e., maximize quantity of attributed apps and quality of apps attributed correctly. High AA with low CA are no much help, since apps are attributed incorrectly.

$$AA = \frac{Apps_attributed}{Total_apps_reviewed} \tag{4.6}$$

$$TCA = \frac{Apps_correctly_attributed}{Total_apps_reviewed} \tag{4.7}$$

$$CA = \frac{TCA}{AA} \tag{4.8}$$

4.4 Reference profiles validation.

To validate the proposed approach, we employed our manually labeled *legitimate authors dataset* and performed 5-fold cross-validation in all exper-

iments. We evaluate our features divided by groups with different machine learning algorithms: Extremely Random Trees (ERT), Extream Gradient Boosting (XGBoosting), AdaBoost, Gradient Boosting, Random Forest, Support Vector Machines (SVM). Results are shown in Table 4.7. Each group of features includes the size of feature vector and the time required to perform computations to create it. As the results show **Random Forest** and **Extremely Random Trees** outperform the rest of the classifiers with all features group, and non-dex related features group. Those classifiers were faster and with better evaluation metrics results.

We repeat this experiment with only *malicious authors dataset*. Results were similar in terms of group of features, however, GradientBoosting outperformed Random Forest this time. Table 4.8 shows only all features group and non dex-related features group.

Finally we combine both previous mentioned datasets and perform the same experiment to decide on the classifier algorithm and features. Table 4.9 shows these results. Again **Random Forest** and **Extremely Random Trees** outperform the rest of the classifiers.

At this point our best features sets are "all features" and "non dex-related features". The second set is a subset of "all features" set that offers excellent results, but it is biased from our initial authors' dataset. Also, this set of features offers less information for a future deep analysis over the apps. Analyzing overall results for all algorithms, "all features" set offer better results, thus this set is selected for next experiments.

In terms of algorithms, **Random Forest** and **Extremely Random Trees** have similar results in terms of accuracy, precision, recall and time required for training and testing our dataset. Both are good candidates for further experiments.

The goal of general ensemble methods is to combine the predictions of several base estimators to improve generalizability and robustness. Random forests algorithm, build several trees in the ensemble using sample drawn with replacement from the training set. When splitting a node during the construction of the tree, the split is not the best among all features, but the best split among a random subset of features. As a result of this randomness, the bias of the forest usually slightly increases with respect to the bias of a single non-random tree but, due to averaging, its variance also decreases, usually compensating more for the increase in bias, yielding an overall better model. In extremely randomized trees, randomness goes one step further in the way splits are performed. A random subset of features is used also, but instead of looking for the best split among them, features are drawn at random to generate the splitting. This randomness tends to reduce the variance.

Because the concern of that additional randomness introduced in Extremely Random Trees could affect results for large scale experiments. We decide to use Random Forest algorithm for next round of experiments.

Table 4.7: Validation on legitimate authors dataset.

Algorithm	**Trainning**	**Testing**	**Accuracy**	**Precision**	**Recall**
All features group , (37,455 features) , (771. secs overall)					
ERT	1.494s	0.167s	**99.9** %	**99.9** %	**99.9** %
XGboosting	644.685s	0.399s	99.6 %	99.2 %	98.9 %
AdaBoost	112.282s	0.751s	73.7 %	57.9 %	59.7 %
GradientBoosting	89.606s	0.067s	99.7 %	99.7 %	99.6 %
Random Forest	1.672s	0.165s	**99.9** %	**99.9** %	99.8 %
SVM	915.159s	24.227s	21.6 %	44.8 %	18.7 %
Direct extracted features group , (189 features) , (9.62 secs overall)					
ERT	0.575s	0.110s	94.7 %	94.6 %	93.4 %
XGboosting	7.792s	0.042s	95.0 %	94.5 %	93.5 %
AdaBoost	0.824s	0.023s	41.5 %	16.7 %	20.9 %
GradientBoosting	22.879s	0.017s	34.5 %	33.7 %	33.8 %
Random Forest	0.561s	0.110s	94.1 %	93.6 %	92.2 %
SVM	5.625s	0.127s	8.9 %	13.2 %	6.8 %
Non dex-related features group , (4,246 features) , (11.8 secs overall)					
ERT	0.674s	0.118s	**99.9** %	**99.9** %	**99.9** %
XGboosting	103.811s	0.086s	99.7 %	99.3 %	99.1 %
AdaBoost	7.614s	0.094s	69.7 %	51.3 %	53.5 %
GradientBoosting	17.094s	0.012s	**99.9** %	**99.9** %	**99.9** %
Random Forest	0.743s	0.118s	**99.9** %	**99.9** %	**99.9** %
SVM	49.510s	1.358s	19.6 %	36.8 %	18.6 %
StrucSignatures features group , (608 features) , (7.72 secs overall)					
ERT	0.513s	0.110s	89.7 %	87.6 %	87.6 %
XGboosting	25.226s	0.047s	90.8 %	88.5 %	88.1 %
AdaBoost	0.914s	0.030s	34.7 %	9.8 %	16.7 %
GradientBoosting	29.444s	0.023s	1.7 %	1.5 %	2.0 %
Random Forest	0.564s	0.111s	90.1 %	88.2 %	87.6 %
SVM	1.772s	0.062s	78.4 %	81.3 %	72.9 %
All opcodes features group , (16,408 features) , (283. secs overall)					
ERT	1.202s	0.146s	94.3 %	94.4 %	92.9 %
XGboosting	349.688s	0.207s	93.9 %	92.8 %	92.4 %
AdaBoost	62.360s	0.351s	37.2 %	17.5 %	18.4 %
GradientBoosting	257.028s	0.042s	89.6 %	90.1 %	87.9 %
Random Forest	1.311s	0.144s	94.7 %	95.2 %	93.9 %
SVM	576.852s	14.830s	71.0 %	85.4 %	65.8 %

Continued on next page

Table4.7 –continued from previous page					
Algorithm	**Trainning**	**Testing**	**Accuracy**	**Precision**	**Recall**
Arrays opcodes features group , (8,019 features) , (77.5 secs overall)					
ERT	1.730s	0.114s	90.3 %	88.8 %	87.7 %
XGboosting	274.152s	0.119s	89.4 %	87.3 %	87.1 %
AdaBoost	11.852s	0.162s	26.3 %	7.3 %	12.3 %
GradientBoosting	471.935s	0.027s	17.8 %	19.5 %	16.9 %
Random Forest	1.227s	0.122s	89.9 %	88.4 %	87.5 %
SVM	1.199s	0.033s	70.8 %	85.2 %	65.0 %
Func. labels from opcodes features group , (8,198 features) , (380. secs overall)					
ERT	1.053s	0.124s	93.6 %	93.5 %	92.1 %
XGboosting	253.390s	0.119s	91.7 %	89.6 %	89.5 %
AdaBoost	26.247s	0.166s	23.7 %	11.1 %	11.8 %
GradientBoosting	176.970s	0.017s	85.5 %	84.8 %	82.5 %
Random Forest	0.839s	0.123s	93.5 %	93.8 %	92.0 %
SVM	15.078s	0.435s	76.1 %	81.6 %	70.8 %
All features except Non dex-related group , (33,209 features) , (853. secs overall)					
ERT	1.551s	0.171s	94.6 %	94.5 %	93.0 %
XGboosting	1101.772s	0.492s	93.7 %	92.7 %	91.9 %
AdaBoost	112.504s	0.748s	32.5 %	13.5 %	15.2 %
GradientBoosting	378.999s	0.065s	89.1 %	90.5 %	86.9 %
Random Forest	1.502s	0.165s	94.5 %	93.9 %	92.7 %
SVM	641.986s	15.928s	72.3 %	84.7 %	67.7 %

Table 4.8: Validation on malicious authors dataset.

Algorithm	**Trainning**	**Testing**	**Accuracy**	**Precision**	**Recall**
All features group , (37,455 features) , (157. secs)					
ERT	0.557s	0.119s	98.6 %	98.8 %	98.2 %
XGboosting	40.789s	0.040s	96.9 %	97.3 %	96.7 %
AdaBoost	18.212s	0.102s	30.0 %	26.6 %	36.9 %
GradientBoosting	14.389s	0.010s	99.1 %	99.3 %	98.8 %
Random Forest	0.609s	0.122s	95.5 %	96.0 %	95.8 %
SVM	19.724s	0.603s	41.5 %	54.0 %	36.2 %
Non dex-related features group , (4,246 features) , (2.94 secs)					
Continued on next page					

Table4.8 –continued from previous page

Algorithm	Trainning	Testing	Accuracy	Precision	Recall
ERT	0.564s	0.108s	**99.9** %	**99.9** %	**99.9** %
XGboosting	5.006s	0.011s	97.3 %	97.6 %	97.2 %
AdaBoost	1.084s	0.019s	26.4 %	21.7 %	30.2 %
GradientBoosting	1.250s	0.002s	99.6 %	99.7 %	99.8 %
Random Forest	0.543s	0.107s	99.1 %	98.6 %	98.8 %
SVM	0.875s	0.025s	53.1 %	52.2 %	46.6 %

Table 4.9: Validation over all authors.

Algorithm	Trainning	Testing	Accuracy	Precision	Recall
All features group , (37,455 features) , (1042 secs)					
ERT	2.437s	0.193s	99.7 %	99.5 %	99.6 %
XGboosting	1324.017s	0.640s	99.3 %	98.8 %	98.9 %
AdaBoost	130.109s	0.865s	53.8 %	31.0 %	32.7 %
GradientBoosting	117.3s	0.077s	99.1 %	99.2 %	98.9 %
Random Forest	1.745s	0.182s	99.6 %	99.4 %	99.4 %
SVM	1247.229s	32.69s	22.5 %	46.9 %	19.6 %
Non dex-related features group , (4,246 features) , (14.4 secs)					
ERT	0.801s	0.122s	**99.9** %	**99.9** %	**99.9** %
XGboosting	142.358s	0.115s	99.5 %	99.3 %	99.2 %
AdaBoost	9.103s	0.109s	54.0 %	31.0 %	32.7 %
GradientBoosting	26.76s	0.015s	99.6 %	99.7 %	99.7 %
Random Forest	0.855s	0.126s	**99.9** %	**99.9** %	**99.9** %
SVM	65.108s	1.734s	21.6 %	39.8 %	21.3 %

4.5 Tuning thresholds

The proposed emerging profile analysis approach relies on four different thresholds for attributing binaries and forming new profiles: *benchmarkThreshold*, *incrementalThreshold*, *outlierlimit*, and *buckets* to group similar apps. We conducted a series of experiments to determine the impact of these thresholds on attributing apps using both legitimate and malicious authors dataset combined. Because traditional metrics for evaluation of classification accuracy did not reveal enough information about how the apps were attributed, we evaluated thresholds in terms of percentage of attributed apps AA, percentage of correctly attributed apps among all considered apps TCA, and a percentage of correctly attributed apps among those that were attributed to some profiles CA (See Section 4.3 for a detailed description).

Experimenting with *benchmarkThreshold* When performing a classification task with an ensemble method, we obtain a probability of a new element to belong to different classes, usually selecting the class with higher probability. Instead of select by default the higher probability value, we decide to use *benchmarkThreshold* to affect the decision whether or not an app is attributed to a known profile (See Algorithm 1 line 9). With higher threshold, the decisions are more conservative i.e, if we select 98 as threshold, apps should get a value above it to be attributed to specific author profile, in other words it means that apps are not attributed to higher prediction probability class unless is more than 98%. We perform a 5-fold cross validation, leaving

10 authors out for each time and calculate the mean of AA, TCA and CA metrics. In this experiment we fixed the rest of the thresholds to the following intuitive values: $incrementalThreshold = 10$ since authors have at least 20 apps we intuitively decide to try half for starters, $outlierlimit = maximum$ to keep a conservative value and *buckets*' edges set to $(70, 50, 30, 10)$ intended to offer a balanced five buckets space. We experiment with values from 30 to 90 with increments of 10 and we add also 98 to test more conservative value.

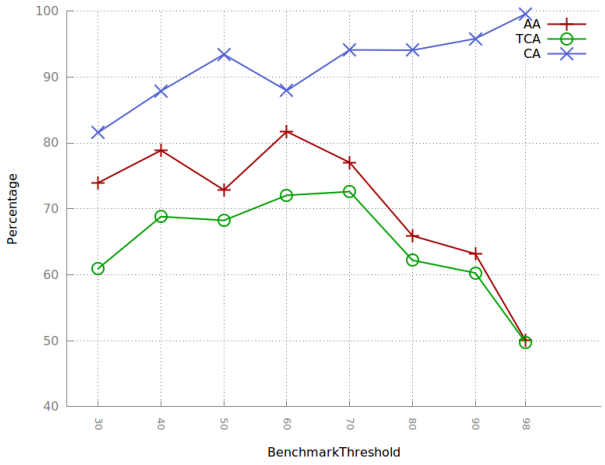

Figure 4.2: Experimental results with various benchmarkThresholds

In Figure 4.2, metrics are shown for values of *benchmarkThreshold*. We want to attribute most of the apps but also attribute them correctly. With 98 as a *benchmarkThreshold*, AA value is the lower result with less than 50% with a CA with the highest value at 100%. The best value for TCA was found at 70 with the best trade off between AA and CA. We use this value of 70 for next experiments.

Experiment with *buckets* edges to group similar apps A key idea to our proposed Algorithm 1 is to group apps based on predicted similarity with classes or profiles we already know. The basic approach is to group together apps assigned to a given class. But we can also divide internal groups from assigned edges, based on $ProbabiltyScore$. The higher this score, the more similarity to a given class. But if the score is so low, chances of similar apps in this group are lower. It is important to properly choose these edges to improve our clustering results. Intuitively, the similarity between apps in the same group decreases if the $ProbabiltyScore$ is lower. If App_1 and App_2 closely resemble App_x, then $ProbabilityScore$ is high and those apps are similar, belonging to the same cluster. In the other hand, if App_4 and App_5 barely resemble App_x, then $ProbabilityScore$ is low and chances of similarity among those apps is uncertain. We experiment with multiple edges; however, we show experimental results only for the following *buckets*' edges to support our selection of them:

a) with no edges at all, one *bucket* to group only by class if $ProbabiltyScore < benchmarkThreshold$.

b) we set a lower edge to 10, we still have only one *bucket* by class.

c) two same size *buckets*. Edges at $(35, 10)$.

d) three same size *buckets*. Edges at $(50, 30, 10)$.

e) four same size *buckets*. Edges at $(55, 40, 25, 10)$.

f) three *buckets* of different size. High $ProbabiltyScore$ lead to bigger *bucket* size. Edges at $(45, 25, 10)$.

g) three *buckets* of different size. High $ProbabiltyScore$ lead to smaller *bucket* size. Edges at $(60, 45, 10)$.

Figure 4.3 shows a graphic representation of the buckets for this experiment. Note that apps below lower edge are not considered belonging to a group. We perform a 5-fold cross validation, leaving 10 authors out for each iteration and calculate the mean of AA, TCA and CA metrics. We fixed the rest of thresholds to following values: $benchmarkThreshold = 70$ as result from previous experiment, $incrementalThreshold = 10$, $outlierlimit = maximum$ as in previous experiment.

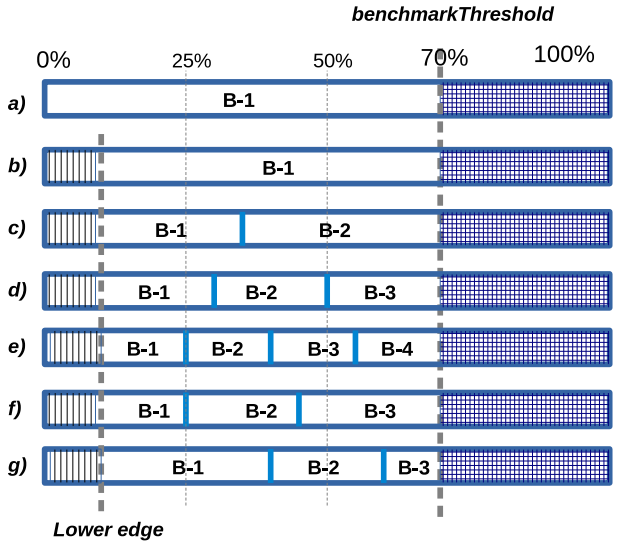

Figure 4.3: Graphical representation of defined bucket edges.

In Figure 4.4, metrics are showed for *buckets*' edges to group similar apps. With only one bucket we achieve higher TCA over 85% but lower CA below 90%. In this case, our framework was attributing apps even before new

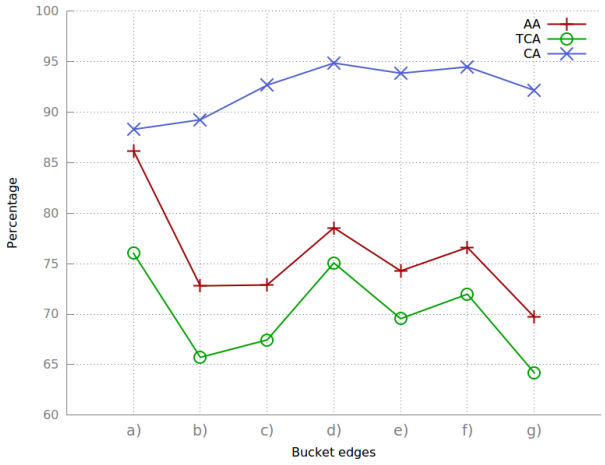

Figure 4.4: Experimental results with defined bucket edges.

profiles were created, so we discard this configuration. The option "d", with edges set at$(50, 30, 10)$ offers the best combined scores in all metrics ($TCA =$ 75.07%, $CA = 94.85\%$ and $AA = 78.55\%$). This configuration was selected for the following experiments.

Experimenting with *incrementalThreshold* This threshold defines the number of apps required to be grouped together in order to form new profiles. In previous experiments we defined 10 apps as default for this threshold, as an intuitive number below actual distribution of apps per author, since our initial group of authors contain at least 20 apps. We experiment with values from 5 to 17 for this threshold. The experiment was performed in a 5-fold cross validation letting 10 authors out each time, with fixed values for the rest of thresholds to the following values: $benchmarkThreshold = 70$ and *bucket* edges setted at$(50, 30, 10)$ as result from previous experiments, and

$outlierlimit = maximum$ as in previous experiments.

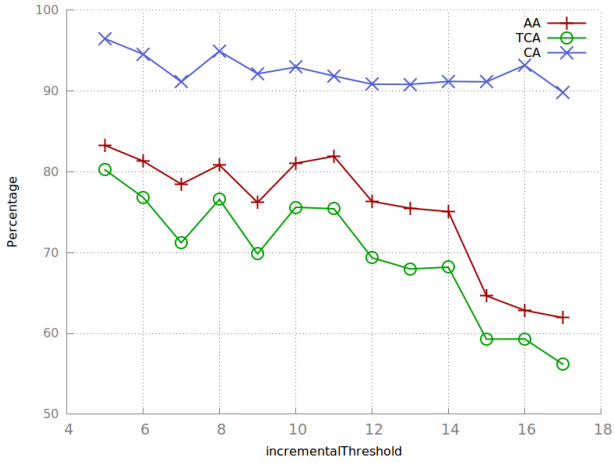

Figure 4.5: Experimental results with various incrementalThresholds.

In Figure 4.5, experimental results for metrics are shown with various values tested. The best results overall are when $incrementalThresholds = 5$ ($TCA = 80.24\%$, $CA = 96.45\%$ and $AA = 83.26\%$). Followed by 8 and 10 apps. Using a small number of apps to create new profiles lets the system generate more new profiles when analyzing unlabeled data, but also increase the complexity of the model increasing the time for training and testing. If we increase this threshold, it is possible to ignore authors with only a few apps in the market to create their profile. A balance between author's discovery and system complexity is desired. This threshold could vary when performing different analysis or investigation with our proposed system.

Experimenting with *outlierlimit* When having large groups of apps we need to detect outliers between them and exclude those from being used to

create a new author profile. These outliers are apps that were grouped in the same bucket but they are different from the rest. In this experiment we test with three approaches: no outlier verification, maximum or minimum value from our outlier detection approach. We perform a 5-fold cross validation letting 10 authors out each time. We fixed the rest of thresholds to following values:$benchmarkThreshold = 70$, *bucket* edges setted at$(50, 30, 10)$ and $incrementalThreshold = 5$ as result from previous experiments.

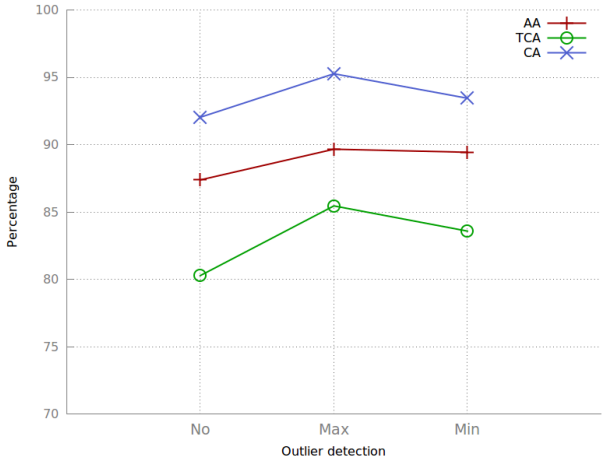

Figure 4.6: Experimental results with different outliers approaches.

In Figure 4.6, metrics for each of three tested approaches are shown. Detecting outliers help us to improve results. Intuitively, using a minimum value between those calculated should lead to increase the CA metric, because it is closing the limits for outliers. In this experiment results, metric values are very similar when using maximum value, expanding the outlier limits. This approach helps to create new profiles while keeping the group cohesive.

The final selected values for the rest of the experiments are fixed to the fol-

lowing values: $benchmarkThreshold = 70$, *bucket* edges setted at$(50, 30, 10)$, $incrementalThreshold = 5$ and $outlierlimit = maximum$.

4.6 Evaluation of emergent-profiles.

In this section we focus on the attribution of unknown apps, we evaluate our approach from different perspectives, using labeled datasets with different characteristics.

Evaluation over labeled dataset In real life, we got only a few profiles to start from all the possible ones. Because our proposed approach had incremental learning capabilities, we perform an experiment to observe what could be a real life behavior. We perform this experiment as a closed world problem, where we know all possible authors beforehand. Each training set contains 10%(5 authors) or 20% (10 authors) from the complete dataset (54 authors). These authors for the training set where chosen in three different ways: a) authors with the largest number of apps, b) authors with the least number of apps, and c) random selection. The remaining authors were used for testing. The provided values for CA, TCA and AA are averaged over 5 runs.

Figure 4.7 presents results for emergent-profile evaluation with different configurations. When we have more authors for training phase we obtain better results (as we can see in a,c,e configurations). Values of calculated metrics

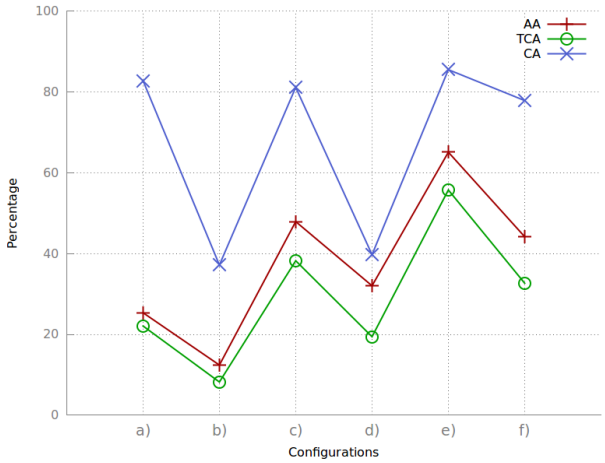

Figure 4.7: Experimental results for emergent-profile evaluation with different configurations. a) 20% authors with the largetst number of apps, b) 10% authors with the largest number of apps, c) 20% authors with the lest number of apps, d) 10% authors with the lest number of apps, e) 20% authors randomly selected and f) 10% authors randomly selected

in the experimental results are below 70%, but they show potential to be accurate when discovering new author profiles. Configuration *e* (20% authors randomly selected) showed the best performance, diversity on the authors and number of apps for each one proved to offer better results than choosing authors with more or fewer apps. A limitation for this experiment is that we stop adding more authors and apps to the system to be tested, thus limiting our approach to discover and create new profiles in subsequent rounds of testing.

From this set of experiments we confirm that the initial set of known authors affects the outcoming results from our authorship attribution framework. As our initial set of known authors offers diversity between them and excellent

Table 4.10: Experimental results from discovery approach considering malware families as coming from same author.

Group	apps	AA	TCA	CA
HackingTeam	128	(128) 100.00%	97.66%	(125) 97.66%
Adware	322	(156) 48.45%	39.13%	(126) 80.77%
Ransomware	472	(378) 80.08%	63.14%	(298) 78.84%
Drebin	3310	(2560) 77.34%	60.24%	(1994) 77.89%

accuracy, we consider it as a good initial set for further experimentation.

Evaluation over malware families. It is important to distinguish malicious authors and malware families. Although they can be related, they are not the same. All malware samples in one family are coming from one or several authors or developers. Similarly, one author could be responsible for several not related malware samples, assigned to different families by AV vendors. These two cases do not exclude each other.

In this experiment we evaluate our discovery approach over a set of malware families, where we consider the first case i.e, that a malware family is created by one or more malicious authors.

To perform this experiment, we created our base classifier with legitimate and malware authors dataset, then we performed our discovery approach over hacking team, adware, ransomware and partial drebin datasets.

Table 4.10 shows a summary results for this experiment. As we mentioned before, although malware families do not necessarily correspond to authors, we still see encouraging results. HackingTeam dataset showed best performance, 101 out of 128 apps were involved in the creation of a new profile,

125 apps were attributed to this profile in next round. Only 3 apps were mixed with other families and attributed to a different profile. No apps were attributed to the HakingTeam profile in following rounds. Adware dataset showed the lowest performance, with only 48% of apps attributed. Out of these 48% only 80% were attributed correctly. A report from Lookout anti-virus company shows that these families are related. In fact, this report describes all these apps as trojanized versions of legitimate apps [1]. This explains low accuarcy as a trojanized version of a legitimate app includes code from original developer and from malicious actor combined, reducing the efficacy of our system while analyzing complete apps. Ransomware and Drebin datasets showed higher accuracy. As we expected, emergent-profiles and attributed apps results were related to malware families.

With these datasets, 80% of apps were attributed with almost 80% of accuracy, considering malware family as same origin. Adware, ransomware and Drebin group results are expressed as an average of metrics for each origin contained in those groups.

As a note, five families from Drebin dataset were not attributed at all (AccuTrack, Biige, TrojanSMS.Hippo, Placms and Vdloader). All these families had only a few number of samples (less than 11). This fact indicates that we need more than that number of apps from a single author to facilitate creation of a new profile.

Table 4.11: Experimental results for emergent-profiles considering malware families as coming from same author. (First Part).

Family name	Apps	Pure P.	Mixed P.	P	C	Other profiles	CA	iCA	i P	Profiles
HackingTeam	128	1 (101)	1 (3)	1	1	fakeDefender	125	3	1	fakeDefender
kemoge	87	1 (22)	7 (33)	3	4	Plankton, shuanet, GinMaster, ransombo	47	15	4	EFA5C21. . . 20062, Plankton, shuanet, ransombo
shedun	83	3 (44)	8 (13)	3	5	kemoge, shuanet, Plankton, GinMaster, ransombo	57	14	6	kemoge, ransombo, shuanet, 4d6c868b, Plankton, GinMaster
shuanet	24	0 (0)	2 (22)	1	1	pletor	22	1	1	pletor
fakeDefender	44	0 (0)	3 (44)	1	2	slocker, Plankton	41	3	2	slocker, Plankton
koler	74	0 (0)	6 (35)	3	3	Mobilespy, pletor, slocker	61	7	3	Mobilespy, pletor, slocker
pletor	15	0 (0)	2 (11)	1	1	svpeng	10	3	2	svpeng, 4d6c868b
ransombo	100	0 (0)	2 (100)	1	1	svpeng	56	44	1	svpeng
slocker	70	2 (23)	10 (16)	3	6	svpeng, koler, ransombo, pletor, DroidKungFu, fakeDefender	32	13	6	svpeng, 4d6c868b, ransombo, pletor, DroidKungFu, fakeDefender
svpeng	100	0 (0)	5 (94)	2	2	koler, pletor	98	10	2	koler, pletor
AccuTrack	10	0 (0)	0 (0)	0	0		0	0	0	
Adrd	74	0 (0)	7 (44)	4	2	Geinimi, Glodream	45	5	2	Geinimi, Glodream
BaseBridge	129	1 (39)	21 (76)	4	6	MobileTx, Glodream, Plankton, DroidKungFu, DroidDream, GinMaster	71	38	6	Glodream, MobileTx, Adrd, DroidKungFu, DroidDream, GinMaster
Biige	10	0 (0)	0 (0)	0	0		0	0	0	
Boxer	17	0 (0)	4 (9)	1	3	SMSreg, FakeInstaller, Iconosys	8	3	2	SMSreg, Iconosys
Copycat	12	0 (0)	4 (9)	0	2	GinMaster, FakeRun	0	8	1	GinMaster
Cosha	11	0 (0)	1 (1)	0	1	Geinimi	0	1	1	Geinimi
Dougalek	17	0 (0)	1 (17)	1	0		17	0	0	
DroidDream	79	0 (0)	11 (53)	1	4	Geinimi, BaseBridge, DroidKungFu, GinMaster	48	8	3	BaseBridge, Geinimi, GinMaster
DroidKungFu	546	2 (23)	40 (409)	18	10	BaseBridge, MobileTx, Kmin, Geinimi, Opfake, Adrd, Plankton, SendPay, Iconosys, GinMaster	419	68	10	BaseBridge, MobileTx, Kmin, Iconosys, Geinimi, Opfake, Adrd, Plankton, GinMaster, SendPay
ExploitLinuxLotoor	61	1 (10)	12 (25)	2	9	SMSZombie, BaseBridge, FakeTimer, Xsider, FakeInstaller, Adrd, Plankton, Opfake, GinMaster	18	18	10	FakeTimer, SMSZombie, BaseBridge, f2a73396bd38767a, Xsider, FakeInstaller, GinMaster, Plankton, Opfake, Adrd
FakeDoc	42	0 (0)	15 (38)	0	7	BaseBridge, Plankton, Geinimi, GinMaster, Xsider, DroidKungFu, DroidDream	0	29	5	DroidDream, DroidKungFu, GinMaster, Xsider, Plankton
FakeInstaller	164	0 (0)	18 (120)	4	8	SMSZombie, SMSreg, Hamob, Boxer, Opfake, Typstu, Dougalek, Iconosys	35	99	10	SMSZombie, SMSreg, Hamob, Boxer, Opfake, 4d6c868b, Typstu, Dougalek, Iconosys, Adrd
Fakelogo	9	0 (0)	1 (8)	0	1	Plankton	0	8	1	Plankton
Fakengry	12	0 (0)	2 (4)	0	2	Mobilespy, Adrd	0	5	2	Mobilespy, Adrd
FakePlayer	10	0 (0)	4 (7)	0	3	BaseBridge, Opfake, Iconosys	0	8	3	BaseBridge, f2a73396bd38767a, Opfake
FakeRun	61	0 (0)	6 (50)	1	2	GinMaster, Plankton	9	51	2	GinMaster, Plankton

Table 4.12: Experimental results for emergent-profiles considering malware families as coming from same author. (Second Part).

Family name	Apps	Pure P.	Mixed P.	P	C	Other profiles	CA	iCA	i P	Profiles
FakeTimer	12	0 (0)	4 (10)	1	2	Hamob, Yzhc	6	3	1	Hamob
Fatakr	17	0 (0)	2 (14)	0	1	Plankton	0	14	1	Plankton
Gappusin	57	0 (0)	20 (28)	0	10	BaseBridge, SendPay, Kmin, Geinimi, Adrd, Plankton, ExploitLinuxLotoor, DroidKungFu, GinMaster, Glodream	0	14	6	BaseBridge, Kmin, Adrd, Plankton, DroidKungFu, Glodream
Geinimi	76	0 (0)	11 (50)	3	4	Mobilespy, MobileTx, Adrd, Xsider	29	24	2	Mobilespy, Adrd
GinMaster	329	1 (12)	21 (294)	16	3	DroidKungFu, Kmin, Plankton	254	7	3	DroidKungFu, Kmin, Plankton
Glodream	68	0 (0)	11 (41)	2	4	Geinimi, DroidKungFu, GinMaster, Adrd	17	22	4	Geinimi, DroidKungFu, GinMaster, Adrd
Hamob	22	0 (0)	2 (11)	2	0		19	0	0	
Iconosys	152	0 (0)	6 (135)	5	1	FakeInstaller	145	3	2	FakeInstaller, Opfake
Imlog	10	0 (0)	6 (7)	0	3	Hamob, Geinimi, Opfake	0	5	3	Hamob, Geinimi, Opfake
Jifake	15	0 (0)	5 (13)	0	4	Dougalek, Typstu, FakeInstaller, Plankton	0	15	4	Dougalek, Typstu, FakeInstaller, Plankton
Kmin	85	0 (0)	8 (54)	1	6	MobileTx, Glodream, Xsider, DroidKungFu, BaseBridge, Adrd	70	8	5	BaseBridge, DroidKungFu, Glodream, MobileTx, Xsider
Mobilespy	14	0 (0)	3 (12)	2	1	Hamob	13	1	1	Hamob
MobileTx	63	0 (0)	1 (16)	1	0		57	0	0	
Nandrobox	13	0 (0)	2 (3)	0	2	GinMaster, Glodream	0	3	2	GinMaster, Glodream
Nickspy	11	0 (0)	1 (2)	0	1	Geinimi	0	2	1	Geinimi
Nyleaker	18	0 (0)	10 (18)	0	3	Geinimi, Glodream, Plankton	0	11	2	Geinimi, Plankton
Opfake	117	0 (0)	8 (83)	4	3	Typstu, SMSZombie, Plankton	95	17	3	Typstu, 4d6c868b, Plankton
Penetho	11	0 (0)	2 (2)	0	2	Adrd, Xsider	0	5	3	shuanet, Adrd, Xsider
Placms	12	0 (0)	3 (3)	0	1	GinMaster	0	0	0	
Plankton	624	4 (84)	23 (538)	15	3	DroidKungFu, GinMaster, Steek	539	23	3	DroidKungFu, GinMaster, Steek
SendPay	29	0 (0)	3 (24)	1	1	DroidKungFu	19	6	1	DroidKungFu
SMSreg	35	0 (0)	8 (18)	1	4	BaseBridge, SMSZombie, GinMaster, Plankton	10	11	3	BaseBridge, SMSZombie, Plankton
SMSZombie	10	0 (0)	1 (10)	1	0		10	0	0	
Steek	14	0 (0)	2 (13)	1	1	Plankton	13	1	1	Plankton
TrojanSMS.Hippo	13	0 (0)	2 (2)	0	1	DroidKungFu	0	0	0	
Typstu	14	0 (0)	2 (14)	1	1	FakeInstaller	13	1	1	FakeInstaller
Vdloader	12	0 (0)	4 (10)	0	1	GinMaster	0	0	0	
Xsider	14	0 (0)	2 (9)	1	1	Geinimi	8	6	2	Geinimi, d20995a79c0daad6
Yzhc	29	0 (0)	3 (14)	1	2	Mobilespy, FakeTimer	7	6	2	Mobilespy, FakeTimer
Zitmo	14	0 (0)	3 (9)	0	2	BaseBridge, FakeInstaller	0	9	2	BaseBridge, FakeInstaller

In Table 4.11 and Table 4.12 we present detailed results for this experiment. Each table contains the following columns: *Family name*, *Pure P.* shows the number of pure profiles created for this family, that means no samples from other families were falsely identified and grouped with this family, and the number of apps involved. *Mixed P.* shows the number of profiles created from mixed families where apps from this one participate, plus the number of apps involved. Some times different families create a new profile, contributing with different number of apps. *P* shows the number of profiles related to this family, it includes pure profiles and mixed profiles where majority of apps belong to this family. *C* shows the number of profiles where apps from this family participated. *Other profiles* include main family names for mixed profiles where this apps participated. *CA* presents the number of correctly attributed apps to its family profile. *i CA* presents the number of apps attributed to other profiles, *i P* the number of other profiles which apps were predicted and *Profiles* shows the name of those other profiles.

To understand significance of the obtained results besides HackingTeam, lets consider the following case where we discovered that families Plankton, Fatakr and Nyleaker are related. Plankton family generated 4 pure profiles with 84 apps, this means only apps identified as plankton where used. It also helped to generated other 23 combined profiles with 584 apps, where apps where mixed with other 21 families. When apps from Plankton family in a mixed profile were majority, we label that profile as Plankton related. In total 15 author profiles related with Plankton were created, those include

the 4 pure profiles. From this family, 539 out of 624 apps were correctly attributed to any of this Plankton author profiles, 23 apps were attributed to other profiles. Fatakr family did not generate any related profile, but it mixed with Plankton and Nyleaker to generate mixed profiles. In total 14 out of 17 apps from this family were attributed to Plankton profiles, the rest 3 apps were not attributed. Nyleaker family did not generate any related profile, but mixed with other families to generate mixed profiles. Out of 18 app, only 11 were attributed, 10 to Plankton profiles and one to Geimini profile. The rest seven apps were not attributed. Based on these results from our system, we inferred that those families are related, or share common origins. Apps labeled as Plankton include a stealthy framework for advertising, and contain suspicious functionalities. Those functionalities include botnet behavior, sending premium SMS, device information leakage over http and external code loading [76]. Fatakr is described as fraudulent app advertisement and its functionalities include leakage of device information and sending SMS. [123]. Nyleaks family apps contain functionalities to leak privacy information from an infected device. [78].

Evaluation over legitimate apps. Once we discovered and created author profiles from suspicious apps in the previous experiment, we evaluated datasets containing legitimate apps in this experiment. The objective is to determine if any of those apps are related with suspicious author profiles. We call suspicious author profiles those profiles that include malware au-

Table 4.13: Experimental results from discovery approach over legitimate apps.

Group	apps	profiles	attributed	attributed to malicious
F-droid	2174	112	1155	0
GooglePlay2015	5600	673	3014	1 (hamob related)
GooglePlay2016	1708	87	594	0

thors, and malware families.

To accomplish that, we feed our system with legitimate apps from F-droid, Googleplay2015 and Googleplay2016 datasets. Because the lack of ground truth for these datasets we did not calculate AA, TCA and CA metrics. We only counted the number of new created profiles for each dataset and the apps attributed to any suspicious author profile.

Only one legitimate app was attributed to a suspicious author profile at this time. Later it was confirmed to be identified as Hamob malware by VirusTotal services.

4.7 Experimenting with attribution only approach.

For this set of experiments we do not perform discovery and creation of new profiles. We employed our proposed system to only perform attribution to our created profiles in previous experiments.

Figure 4.8: Malware analysis results from more than three years ago in VirusTotal service.

Authorship attribution over a large unknown stream of suspicious apps. The objective for this experiment is to intent authorship attribution for a large stream of around 40,000 unknown suspicious apps obtained from Virustotal stream service. This service provided us with more than 100,000 Android related files submitted by users to the VirusTotal online service. From all those files about 38,000 were valid apks, the rest of them were .dex files, or other type related with Android by VirusTotal system. These apk files were unlabeled, only after having been attributed we manually label them for comparison purposes. To evaluate our system we analyzed three facts from these apps: a) attributed apps, b) attributed apps to a suspicious author profile created from previous experiments, and c) comparing AV labels for those attributed apps.

Only 33,656 valid apks were considered for authorship attribution process. From those apps, 2,854 (8.48%) were attributed to 88 previously known author profiles, benign and suspicious. From 28 suspicious author profiles cre-

Table 4.14: Experimental results for authorship attribution over a large unknown suspicious stream of apps.

Dataset	apps analyzed	attributed (profiles)	suspicious (profiles)
VirusTotal Stream	33,656	2,854 8.48% (88)	1,465 (28)

ated by malware apps, we identify suspicious 1,465 apps related to them. From these 1,467 apps attributed to suspicious profiles, only four of them had zero detections in VirusTotal system. Usually VirusTotal system shows latest analysis result, which could be minutes or years ago. See Figure 4.8 for an example. We submitted these four files to be analyzed again by the VirusTotal system. We found that two of these four were now detected as unwanted software for at least one antivirus engine, showing that the origin style could lead to detect unknown threats.

Authorship attribution in the Markets. The objective for this experiment is to perform authorship attribution in the wild, analyzing over 43,000 apps collected directly from different Android markets. For these apps we analyzed only to facts: a) attributed apps per market and b) author profiles per market. Suspicious author profiles include malicious authors and malware families.

Only 6.35% of apps were attributed to 89 unique author profiles. From Slideme market zero apps were attributed. An explanation is that authors from this market do not publish their apps in other markets; hence, we do not

Table 4.15: Experimental results for authorship attribution over a large unknown stream of apps.

Dataset	apps analyzed	apps attributed	unique profiles
GooglePlay	40,744	(2,031) 4.98%	54
360	1,159	(214) 18.46%	26
anruan	6,294	(22) 0.35%	8
anzhi	16,063	(2,863) 17.82%	29
gfan	2,451	(26) 1.06%	5
slideme	6,340	(0) 0.00%	0
tencent	6,068	(160) 2.64%	15
xiaomi	5,064	(30) 0.59%	8
total	84,183	(5,346) 6.35%	89

have a profile for those developers. None of the apps analyzed in the third-party markets was attributed to a suspicious profile; this does not mean that all the apps are safe and clean, but that we do not know dubious authors from those markets. However, from the official GooglePlay market, 52 apps were attributed to three different suspicious author profiles in groups of 1,2 and 49 apps. An analysis of the group of 49 apps in Virustotal, reported 22 apps as non-suspicious from original analysis dated between one year and a half and several months ago. After submitting again those apps to be analyzed, they were reported now as suspicious at least by one anti-virus. Another example of promising results from our framework in this experiment is that two groups of apps created by same developing tool (appinventor and app4mobile) where

attributed to their respectively profiles.

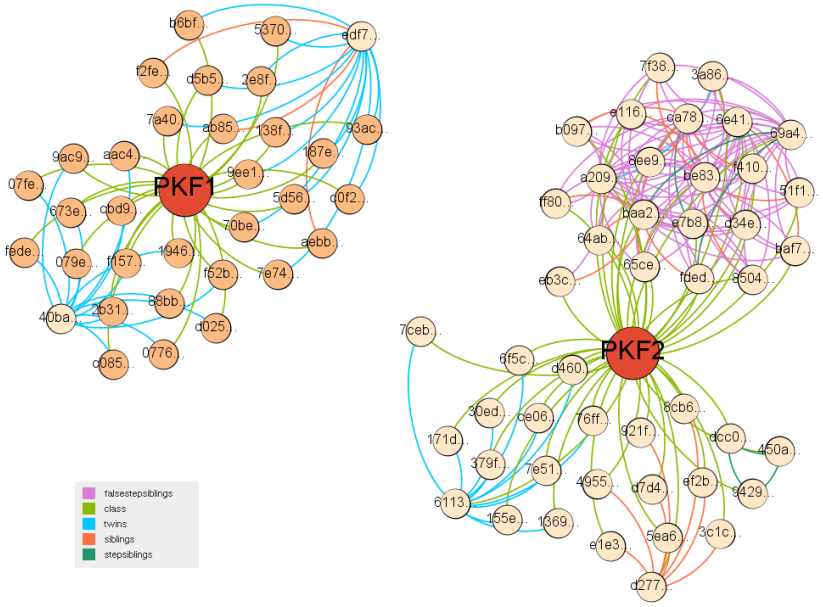

Figure 4.9: Droidkin results for the two larger groups of apps attributed to DroidKungfu malware related profiles.

Authorship attribution over a stream of known malicious apps. In this experiment, our objective is to perform authorship attribution for all the apps belonging to the Drebin dataset, including the families with less than 20 apps discarded for previous experiments. Using the labels already offered as ground truth, we analyzed the number of attributed apps, attributed families, and author profiles created. We also offer two new sets of labels for attributed apps, one based on author profiles attributed to, and the second one based on family and author profile attributed to. With these new sets of labels as ground truth, we expect to help the community to perform further

research on authorship attribution on Android malware.

For this experiment, we employ one model created from malware apps, i.e., Drebin families, adware, ransomware, HackingTeam datasets; plus legitimate apps from F-droid, Google2015, and Google2016 datasets. This model contains 1019 author profiles, which includes ten malicious authors from reference profiles plus 87 profiles created from the malicious dataset. The rest 879 author profiles were created from legitimate apps. From experimental results, the total number of attributed apps is 3,442 (62.12%) out of 5,541. From these attributed apps, 130 (3.78%) were attributed to non-suspicious profiles. During the training phase, we only use 47 Drebin families with 3181 apps. While performing attribution, apps from 54 families were intended to attributed to different author profiles. From the attributed apps, only ten of them belonged to these seven new families, one app of each family was attributed to an existent profile as shown in Table 4.16. The other three were attributed to non-suspicious profiles. We check manually those seven apps on VirusTotal to verify findings from our system, corroborating relations between malicious authors profiles created from families and malware classification from anti-virus products.

We found that analyzed apps where attributed to 82 author profiles. Choosing author profiles with more than 10 apps attributed we ended up with only 45 of them with a total of 3304 apps. These author profiles contain apps mixed from different families based on the original labeling. When taking malware family labels into account, we obtain a combination of 223 groups

Table 4.16: Experimental results from attribution over new families' apps.

App	Drebin Family	Malicious profile(s) attributed	Family assigned from VT
729970ae0 . . . 3d2de28e	Loicdos	Iconosys, FakeInstaller	LoicDos, Fakeapp, FakeFlash
477d6be36 . . . 5cd7892	RuFraud	FakeInstaller	FakeInstaller, RuFraud
a65149660 . . . 33f461e	EWalls	Iconosys	EWalls, imlog
c1cefe44f . . . 9942cd7	PJApps	Geinimi, ransombo	PJApps
3ef2dcd86 . . . 445b64b	Saiva	Opfake, koler	Opfake, FakeInstaller, Saiva
09be2dd3b . . . 8e53b33	TrojanSMS-Boxer.AQ	FakeInstaller, MobileTx	Boxer, FakeInstaller
c9cb37e99 . . . 4ecd6f1	Tapsnake	Xsider	Tapsnake

based on family label and author profile, where several malware families were breaking into smaller groups of author profiles. If we regroup these apps, we obtain 67 new groups with more than 10 apps per author, with a total of 2952 apps selected. For the list of apps and families consult online `https://github.com/hugo-glez/authorsdrebian`.

Malware researchers often depend on anti-virus labels when performing any type of classification. In this field is well known that different anti-virus programs could label same malicious sample with different names; hence, assigning author profiles to the samples represents a new way to group them based on stylistic similarity detected by our framework. As an example, in the DroidKungfu malware family there is a small group of samples with only malicious purposes, also there are other groups which seems like malicious code was added to legitimate apps. These groups represent different origins

for our purposes. Those groups were attributed to different profiles by our system. To verify accuracy on our system related to these groups, we compare the two larger groups with Droidkin to identify any possible relationship between the apps belonging to these groups, and we also conduct manual analysis on the apps. Figure 4.9 shows two independent groups of apps with strong relationships. Those apps are not related between groups. In manual analysis we discover two certificates were used to sign apps from each group. Apps in the group share similar advertisement libraries and some urls besides the stylistic similarity detected by our system.

Figure 4.10: A preview of the use case system.

Use case: Evaluating the threat landscape in official Android market. We propose a real use case scenario for our system, where attribution to suspicious profiles is used to evaluate the threat landscape in the official Android market. In this system deployment, we employ a model containing only suspicious profiles created from malicious applications. An ongoing

collection of Android malware is performed to feed our system to discover and create more suspicious profiles. In this case, our system continuously will be growing with only suspicious author profiles. To evaluate the threat landscape, we need to collect newly popular Android applications from the official Market each day. We then perform attribution process to identify suspicious apps. If an app is attributed to a suspicious author profile, we consider that app as suspicious and increase the threat level for the day. The threat level is expressed in GREEN if none of the apps were attributed that day, YELLOW if some apps are attributed, and RED if all apps are attributed for that day. Figure 4.10 shows how the system looks. For a live view of this use case visit: `http://iscxm36.cs.unb.ca/previewTL/`.

4.8 Concluding remarks

With these experiments, we showed that we were developing and creating our model with the best results for each phase. We were carefully looking at models and profiles as they were evolving. We evaluated clear measures when ground truth existed, and we performed manual investigations when results showed interesting information.

Writing a malware detection signatures for each malware sample or malware family encountered is a cumbersome and time-consuming process. Hence, to combat malware effectively, it is desirable to identify malicious authors by grouping malware with strong stylistic similarities, allowing us to write

author profile signatures which capture the commonalities of all malware samples from an author.

During our experiments, we created author profiles automatically by discovering groups of apps with stylistic similarities and let our machine learning method learn new classes for those new groups. We employed our system with real data from different markets to discover new author profiles and to attribute apps; when those apps were assigned to suspicious author profiles, we conduct manual investigations to corroborate our findings. Our system attributed to suspicious profiles several apps from the official Android market, 30 of those apps were considered non-suspicious by Virustotal service when they were evaluated for the first time, but now they are considered suspicious, meaning that antivirus analyzed those samples and updated their signatures. This fact tells us that our system can discover suspicious apps by attributing them to suspicious author profiles.

Chapter 5

Conclusions and future work

"Life is the art of drawing sufficient conclusions from insufficient premises."

Samuel Butler

Open world problems in authorship attribution have been around for a couple of years, gaining more attention recently but with no general solution yet. Typical studies in authorship attribution have their focus on a specific set of authors, leaving out the possibility to attribute unknown authors. On the other hand, machine learning problems can deal with unlabeled datasets offering new opportunities for semi-supervised learning, thus presenting new challenges.

In this thesis, we proposed and evaluated five groups of features fitted for the task of authorship attribution in Android files. Our system extracts the features using static analysis techniques, mainly from .dex files, which we consider the binary container of the app. The features are represented at

different stylistic levels, and they depend on author's preferences. Features chosen showed excellent results in closed world problem to attribute known authors, with 99% of accuracy. We also proposed and evaluated a method to work with an open world problem in authorship attribution; with this method, we obtain acceptable results near 80% of attributed Android apps accuracy with unlabeled datasets.

We faced intellectual and technical challenges; however, we were able to overcome them with an operational framework to discover new probable authors and group them based on stylistic features. From an unknown Android app, we are not able to point fingers to a very specific developer with name and address at the moment but to point to a profile candidate created by our framework. This candidate profile is a step forward to link an Android app with an actual group of specific person.

This framework is based on the modular architecture proposed, it includes our algorithm to discover and create new profiles, and it incorporates modules to evaluate performance on proposed features. Although this framework is at its early stage, it is practical, functional, and scalable.

The datasets for experimentation could become old quickly in the computer security domain. This work will not be an exception. The evolution and updates of the Android operating system, the Android ecosystem, developer tools and the malware itself request for update actions related to the datasets employed to prepare our framework. Datasets to create "common code" should be updated when new major versions of Android framework are

released. As a consequence, the thresholds for our algorithm can be updated at the same time following the steps described in Section 4.5, however is hard to foreseen significant changes if the input dataset is the same.

The original input dataset to create *reference profiles* plays a crucial role in the system to achieve our goals, it represents the seed to our algorithm to discover and create new author profiles from stylistic similarities. If this dataset is replaced, modified or updated, the thresholds for our proposed algorithm should be reexamined. For this purpose, one can start with the actual defined thresholds and then follow the presented steps to verify or update them. The reference profiles represent authors or groups even if we do not know all the whereabouts about them. The first input dataset should be a trustable group of apps from known authors, which are used to create new author profiles later in the process. It is possible to increase or specialize this first dataset for special use cases of the system, like focusing on authors profiles targeting a company or a country, or focusing on state-sponsored actors. We released two Android author datasets, one focus on legitimate authors found in the markets, and other focus on malicious authors with apps intalled on Android phones detected.

Related to experimental results findings, we can state that for some malware families different authors or methods to produce them were employed. Throughout our experiments, we noticed that *HackingTeam* dataset created a single candidate author profile. We also noticed that other families always create several profiles indicating different origins among them, this is

the case for the *DroidKungfu* family. Android apps labeled as DroidKungfu usually are divided into two groups: a) apps with only malicious code, and b) legitimate apps with appended malicious code; this fact shows at least two different origins from this malware. Manual investigations should be conducted to find more relevant information about those apps attributed beyond the profile assigned.

By creating author's profiles instead of malware family signatures, we can keep pace with the evolving landscape of malicious adversaries, malware distribution channels and techniques, and sophisticated groups profiting from insecure systems and users.

Opportunities of future work The first opportunity to extend this work is to add support for different type of files. Our research focus was on Android apps; following a similar trail of thoughts one can extract and represent features from other types of files to be used with the actual framework, as such as iOS apps, Windows binaries or Linux elf format.

The second opportunity is to collect automatically open-source intelligence (OSINT) for the group of apps used to create a new candidate profile; until now this work should be conducted manually. Sometimes by searching related news for a malware sample, other times by analyzing urls or resources in the group.

Finally, the framework contains a modular architecture; therefore, parts of it could be used, extended or improved for other projects. Also, other projects

can be added to the framework.

Bibliography

[1] *Lookout discovers new trojanized adware; 20k popular apps caught in the crossfire*, `https://blog.lookout.com/blog/2015/11/04/trojanized-adware/`, [Online; accessed January 19, 2016].

[2] *Virustotal malware intelligence services*, `https://www.virustotal.com`, [Online; accessed Jan 22, 2017].

[3] Y. Aafer, W. Du, and H. Yin, *DroidAPIMiner: Mining API Level Features for Robust Malware Detection in Android*, Proceedings of the 9th SecureComm (Sydney, Australia), September 25-27 2013.

[4] A.J. Albrecht and J.E. Gaffney, *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*, IEEE Transactions on Software Engineering **SE-9** (1983), no. 6, 639–648.

[5] Aisha Ali-Gombe, Irfan Ahmed, Golden G. Richard, III, and Vassil Roussev, *Opseq: Android malware fingerprinting*, Proceedings of

the 5th Program Protection and Reverse Engineering Workshop (New York, NY, USA), PPREW-5, ACM, 2015, pp. 7:1–7:12.

[6] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi, *Oba2: an onion approach to binary code authorship attribution*, Digital Investigation **11** (2014), S94–S103.

[7] A. J. Alzahrani, N. Stakhanova, H. Gonzalez, and A. Ghorbani, *Characterizing Evaluation Practices of Intrusion Detection Methods for Smartphones*, Journal of Cyber Security and Mobility (2014).

[8] B. Amos, H. Turner, and J. White, *Applying machine learning classifiers to dynamic android malware detection at scale*, 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC), July 2013, pp. 1666–1671.

[9] Android, *Dalvik Executable Format*, `http://source.android.com/devices/tech/dalvik/dex-format.html`, [Online; accessed July 22, 2014].

[10] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck, *Drebin: Effective and explainable detection of android malware in your pocket*, Proceedings of the 21th Annual NDSS, 2014.

[11] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario, *Automated classification and analysis of*

internet malware, pp. 178–197, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[12] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji, *A methodology for empirical analysis of permission-based security models and its application to android*, Proceedings of the 17th ACM Conference on CCS '10 (New York, NY, USA), ACM, 2010, pp. 73–84.

[13] Rudolf Bayer, MH Halstead, and Necdet Bulut, *Experimental validation of a structural property of fortran algorithms*, ACM '74 Proceedings of the 1974 annual conference (1974), 207–211.

[14] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan, *Mockdroid: Trading privacy for application functionality on smartphones*, Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (New York, NY, USA), HotMobile '11, ACM, 2011, pp. 49–54.

[15] HL Berghel and DL Sallach, *Measurements of program similarity in identical task environments*, ACM SIGPLAN Notices (1984), no. August.

[16] P. Vinod Bhattathiripad, *Software Piracy Forensics: A Proposal for Incorporating Dead Codes and Other Programming Blunders as Important Evidence in AFC Test*, 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops (2012), 206–212.

[17] Leo Breiman, *Random forests*, Machine learning **45** (2001), no. 1, 5–32.

[18] Necdet Bulut and MH Halstead, *Invariant properties of algorithms*, ACM SIGPLAN Notices (1973), 12–13.

[19] Steven Burrows and S M M Tahaghoghi, *Source Code Authorship Attribution using n -grams*, Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University, 2007, pp. 32–39.

[20] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin, *Comparing techniques for authorship attribution of source code*, Software: Practice and Experience **44** (2014), no. August 2012, 1–32.

[21] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt, *De-anonymizing programmers via code stylometry*, 24th USENIX Security Symposium (USENIX Security 15) (Washington, D.C.), USENIX Association, August 2015, pp. 255–270.

[22] G. Canfora, A. D. Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, *Effectiveness of opcode ngrams for detection of multi family android malware*, Availability, Reliability and Security (ARES), 2015 10th International Conference on, Aug 2015, pp. 333–340.

[23] Dong-Kyu Chae, Sang-Wook Kim, Jiwoon Ha, Sang-Chul Lee, and Gyun Woo, *Software plagiarism detection via the static API call fre-*

quency birthmark, Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13 (2013), 1639.

[24] Checkpoint, *More than 1 million google accounts compromised by gooligan campaign*, `http://blog.checkpoint.com/2016/11/30/1-million-google-accounts-breached-gooligan/`, [Online; accessed Dec 12, 2016].

[25] Seokwoo Choi, Heewan Park, Hyun-il Lim, and Taisook Han, *A static API birthmark for Windows binary executables*, Journal of Systems and Software **82** (2009), no. 5, 862–873.

[26] Radhouane Chouchane, Natalia Stakhanova, Andrew Walenstein, and Arun Lakhotia, *Detecting machine-morphed malware variants via engine attribution*, Journal of Computer Virology and Hacking Techniques (2013).

[27] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster, *Visual reverse engineering of binary and data files*, Visualization for Computer Security, Springer, 2008, pp. 1–17.

[28] CrowdStrike, *Putter panda*, Intelligence Report, June 2014.

[29] Jonathan Crussell, Clint Gibler, and Hao Chen, *Scalable semantics-based detection of similar android applications*, 18th ESORICS (Egham, U.K.).

[30] Jonathan Crussell, Clint Gibler, and Hao Chen, *Attack of the clones: Detecting cloned applications on android markets*, 17th ESORICS), vol. 7459, Springer, 2012, pp. 37–54.

[31] B. Curtis, S.B. Sheppard, P. Milliman, M.a. Borst, and T. Love, *Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics*, IEEE Transactions on Software Engineering **SE-5** (1979), no. 2, 96–104.

[32] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and C Seifert, *Zozzle: Low-overhead mostly static JavaScript malware detection*, Proceedings of the Usenix . . . (2011).

[33] G Data, *G data mobile malware report*, White Paper, April 2015.

[34] E. R. de Faria, I. R. Gonalves, J. Gama, and A. C. P. d. L. F. Carvalho, *Evaluation of multiclass novelty detection algorithms for data streams*, IEEE Transactions on Knowledge and Data Engineering **27** (2015), no. 11, 2961–2973.

[35] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward, *Hints on test data selection: Help for the practicing programmer*, Computer **11** (1978), no. 4, 34–41.

[36] Anthony Desnos, *Android: Static Analysis Using Similarity Distance*, 2012 45th Hawaii International Conference on System Sciences (2012), no. X, 5394–5403.

[37] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach, *Quire: Lightweight provenance for smart phone operating systems*, Proceedings of the 20th USENIX Conference on Security (Berkeley, CA, USA), SEC'11, USENIX Association, 2011, pp. 23–23.

[38] Haibiao Ding and Mansur H. Samadzadeh, *Extraction of Java program fingerprints for software authorship identification*, Journal of Systems and Software **72** (2004), no. 1, 49–57.

[39] JL Donaldson, AM Lancaster, and PH Sposato, *A plagiarism detection system*, ACM SIGCSE Bulletin (1981), 21–25.

[40] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel, *A survey on automated dynamic malware-analysis techniques and tools*, ACM Comput. Surv. **44** (2008), no. 2, 6:1–6:42.

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth, *Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones*, Proceedings of the 9th USENIX Conference on OSDI'10 (Berkeley, CA, USA), USENIX Association, 2010, pp. 1–6.

[42] William Enck, Machigar Ongtang, and Patrick McDaniel, *On lightweight mobile phone application certification*, Proceedings of the 16th ACM Conference on CCS '09 (New York, NY, USA), ACM, 2009, pp. 235–245.

[43] JAW Faidhi and SK Robinson, *An empirical approach for detecting program similarity and plagiarism within a university programming environment*, Computers & Education **11** (1987), no. 1, 11–19.

[44] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner, *A survey of mobile malware in the wild*, Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (New York, NY, USA), SPSM '11, ACM, 2011.

[45] A Fitzsimmons and T Love, *A review and evaluation of software science*, ACM Computing Surveys (CSUR) **10** (1978), no. 1.

[46] Jeff Forristal, *Android: One root to own them all*, BlackHat, 2013.

[47] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis, *Examining the significance of high-level programming features in source code author classification*, Journal of Systems and Software **81** (2008), no. 3, 447–460.

[48] Georgia Frantzeskou and E Stamatatos, *Identifying authorship by byte-level n-grams: The source code author profile (scap) method*, International Journal of ... **6** (2007), no. 1, 1–18.

[49] Georgia Frantzeskou and Efstathios Stamatatos, *Source code author identification based on n-gram author profiles*, Artificial Intelligence ... (2006).

[50] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas, *Effective identification of source code authors using byte-level information*, Proceeding of the 28th international conference on Software engineering - ICSE '06 (2006), 893.

[51] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck, *Structural detection of android malware using embedded call graphs*, Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (New York, NY, USA), AISec '13, ACM, 2013, pp. 45–54.

[52] Clint Gibler, Ryan Stevens, Jon Crussell, Hao Chen, Hui Zang, and Heesook Choi, *Adrob: Examining the landscape and impact of android application plagiarism*, Proceedings of the 11th MobiSys (Taipei, Taiwan), 2013.

[53] H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, *Measuring code reuse in android apps*, 2016 14th Annual Conference on Privacy, Security and Trust (PST), Dec 2016, pp. 187–195.

[54] Hugo Gonzalez, *Android authorship attribution framework*, https://iscxqr2.cs.unb.ca/hugo-glez/nahuathl, [Online; accessed April 27, 2016].

[55] ———, *Common code source code*, https://iscxqr2.cs.unb.ca/hugo-glez/CommonCodeTool, [Online; accessed April 27, 2016].

[56] ———, *Stringanalysis source code*, https://iscxqr2.cs.unb.ca/hugo-glez/stringanalysis, [Online; accessed April 27, 2016].

[57] ———, *Strucsignature source code*, https://iscxqr2.cs.unb.ca/hugo-glez/structsignature, [Online; accessed April 27, 2016].

[58] ———, *androsoo: First release*, https://doi.org/10.5281/zenodo.155354, May 2015, [Online; accessed Sep 27, 2016].

[59] ———, *droidcolors: Beta for conference*, https://doi.org/10.5281/zenodo.155355, December 2015, [Online; accessed Sep 27, 2016].

[60] ———, *dexstrings: First release*, https://doi.org/10.5281/zenodo.155080, September 2016, [Online; accessed Sep 27, 2016].

[61] Hugo Gonzalez, Andi A. Kadir, Natalia Stakhanova, Abdullah J. Alzahrani, and Ali A. Ghorbani, *Exploring reverse engineering symptoms in android apps*, Proceedings of the Eighth European Workshop on System Security (New York, NY, USA), EuroSec '15, ACM, 2015, pp. 7:1–7:7.

[62] Hugo Gonzalez, Natalia Stakhanova, and Ali Ghorbani, *Droidkin: Lightweight detection of android apps similarity*, Proceedings of the 10th SECURECOMM, 2014.

[63] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang, *Riskranker: scalable and accurate zero-day android malware detection*, The 10th MobiSys, 2012, pp. 281–294.

[64] Sam Grier, *A tool that detects plagiarism in pascal programs*, ACM SIGCSE Bulletin, vol. 13, ACM, 1981, pp. 15–20.

[65] M H Halstead, *Natural laws controlling algorithm structure?*, ACM Sigplan Notices (1972), 19–26.

[66] MH Halstead, *Toward a theoretical basis for estimating programming effort*, Proceedings of the 1975 annual conference (1975), 222–224.

[67] PG Hamer and GD Frewin, *MH Halstead's Software Science-a critical examination*, ... of the 6th international conference on Software ... (1982), 197–206.

[68] Kyoung Soo Han, Jae Hyun Lim, Boojoong Kang, and Eul Gyu Im, *Malware analysis using visualized images and entropy graphs*, International Journal of Information Security **14** (2015), no. 1, 1–14.

[69] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song, *Juxtapp: A scalable system for detecting code reuse among android applications*, Proceedings of the 9th DIMVA'12 (Berlin, Heidelberg), Springer-Verlag, 2013, pp. 62–81.

[70] Jane Huffman Hayes and Jeff Offutt, *Recognizing authors: an examination of the consistent programmer hypothesis*, Software Testing, Verification and Reliability **20** (2010), no. 4, 329–356.

[71] JH Hayes, *Authorship Attribution: A Principal Component and Linear Discriminant Analysis of the Consistent Programmer Hypothesis.*, IJ Comput. Appl. (2008).

[72] Nwokedi Idika and Aditya P Mathur, *A survey of malware detection techniques*, Purdue University **48** (2007).

[73] Ashutosh Jain, Hugo Gonzalez, and Natalia Stakhanova, *Enriching reverse engineering through visual exploration of android binaries*, Proceedings of the 5th Program Protection and Reverse Engineering Workshop (New York, NY, USA), PPREW-5, ACM, 2015, pp. 9:1–9:9.

[74] Hugo T. Jankowitz, *Detecting plagiarism in student pascale programs*, The Computer Journal **31** (1988), no. 1, 1–8.

[75] Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho, *A Plagiarism Detection Technique for Java Program Using Bytecode Analysis*, 2008 Third International Conference on Convergence and Hybrid Information Technology (2008), 1092–1098.

[76] Xuxian Jiang, *Security alert: New stealthy android spyware – plankton – found in official android market*, `https://www.csc2.ncsu.edu/faculty/xjiang4/Plankton/`, [Online; accessed Oct 14, 2016].

[77] Meenu Mary John, P. Vinod, and K. A. Dhanya, *Hartley's test ranked opcodes for android malware analysis*, Proceedings of the 8th International Conference on Security of Information and Networks (New York, NY, USA), SIN '15, ACM, 2015, pp. 304–311.

[78] Juniper, *Mobile signature descriptions*, `https://www.juniper.net/security/auto/includes/mobile\_signature\_descriptions.html\#ANDROID:A.NYLeaker`, [Online; accessed Oct 14, 2016].

[79] B. Kang, S. Y. Yerima, K. Mclaughlin, and S. Sezer, *N-opcode analysis for android malware classification and categorization*, 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), June 2016, pp. 1–7.

[80] Byeongho Kang, BooJoong Kang, Jungtae Kim, and Eul Gyu Im, *Android malware classification method: Dalvik bytecode frequency analysis*, Proceedings of the 2013 Research in Adaptive and Convergent Systems (New York, NY, USA), RACS '13, ACM, 2013, pp. 349–350.

[81] Joris Kinable and Orestis Kostakis, *Malware classification based on call graph clustering*, Journal in Computer Virology **7** (2011), no. 4, 233–245.

[82] Deguang Kong, Yoon-Chan Jhi, Tao Gong, Sencun Zhu, Peng Liu, and Hongsheng Xi, *Sas: semantics aware signature generation for polymor-*

phic worm detection, International Journal of Information Security **10** (2011), no. 5, 269–283.

[83] Koodous, *Collaborative platform for apk analysis*, `https://koodous.com`, [Online; accessed June 22, 2017].

[84] Ivan Krsul, *Authorship Analysis: Identifying the Author of a Program*, Tech. report, Purdue University, 1994.

[85] Ivan Krsul and EH Spafford, *Authorship analysis: Identifying the author of a program*, (1995).

[86] Ivan Krsul and Eugene H. Spafford, *Authorship analysis: Identifying the author of a program*, Computers and Security (1996).

[87] Ivan Krsul and Eugene H. Spafford, *Authorship analysis: identifying the author of a program*, Computers & Security **16** (1997), no. 3, 233–257.

[88] M. La Polla, F. Martinelli, and D. Sgandurra, *A survey on security for mobile devices*, Communications Surveys Tutorials, IEEE **15** (2013).

[89] Kaspersky Lab, *Global corporate it security risks: 2013*, White Paper, May 2013.

[90] Robert Charles Lange and Spiros Mancoridis, *Using code metric histograms and genetic algorithms to perform author identification for*

software forensics, Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07 (2007), 2082.

[91] Robert Layton and Ahmad Azab, *Authorship analysis of the zeus botnet source code*, Cybercrime and Trustworthy Computing Conference (CTC), 2014 Fifth, Nov 2014, pp. 38–43.

[92] Robert Layton, Paul Watters, and Richard Dazeley, *Automatically determining phishing campaigns using the USCAP methodology*, 2010 eCrime Researchers Summit (2010), 1–8.

[93] ———, *Automated unsupervised authorship analysis using evidence accumulation clustering*, Natural Language Engineering **19** (2013), no. 01, 95–120.

[94] Mandiant, *Apt1: Exposing one of chinas cyber espionage units*, White Paper, February 2013.

[95] H. Mekky, A. Mohaisen, and Zhi-Li Zhang, *Separation of benign and malicious network events for accurate malware family classification*, 2015 IEEE Conference on Communications and Network Security (CNS), Sept 2015, pp. 125–133.

[96] Mohammad Nauman, Sohail Khan, and Xinwen Zhang, *Apex: Extending android permission model and enforcement with user-defined runtime constraints*, Proceedings of the 5th ASIACCS '10 (New York, NY, USA), ACM, 2010, pp. 328–332.

[97] PW Oman and CR Cook, *Programming style authorship analysis*, Proceedings of the 17th conference on ACM ... (1989), 320–326.

[98] KJ Ottenstein, *An algorithmic approach to the detection and prevention of plagiarism*, ACM Sigcse Bulletin (1976), 30–41.

[99] a. Parker and J.O. Hamblen, *Computer algorithms for plagiarism detection*, IEEE Transactions on Education **32** (1989), no. 2, 94–99.

[100] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al., *Scikit-learn: Machine learning in python*, Journal of Machine Learning Research **12** (2011), no. Oct, 2825–2830.

[101] N. Peiravian and X. Zhu, *Machine learning for android malware detection using permission and api calls*, 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Nov 2013, pp. 300–305.

[102] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang, *Plagiarizing smartphone applications: Attack strategies and defense techniques*, Proceedings of the 4th ESSoS'12 (Berlin, Heidelberg), Springer-Verlag, 2012, pp. 106–120.

[103] Lutz Prechelt, Guido Malpohl, and Michael Philippsen, *Finding Plagiarisms among a Set of Programs with JPlag*, J. UCS **8** (2002), no. 11, 1016–1038.

[104] Mykola Protsenko and Tilo Müller, *Android malware detection based on software complexity metrics*, pp. 24–35, Springer International Publishing, Cham, 2014.

[105] M. Zubair Rafique, Ping Chen, Christophe Huygens, and Wouter Joosen, *Evolutionary algorithms for classification of malware families through different network behaviors*, Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (New York, NY, USA), GECCO '14, ACM, 2014, pp. 1167–1174.

[106] Paul Rascagneres, *Apt1: technical backstage*, White Paper, March 2013.

[107] D Krishna Sandeep Reddy and Arun K Pujari, *N-gram analysis for computer virus detection*, Journal in Computer Virology **2** (2006), no. 3, 231–239.

[108] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov, *Learning and classification of malware behavior*, pp. 108–125, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[109] Konrad Rieck, Christian Wressnegger, and Alexander Bikadorov, *Sally: A tool for embedding strings in vector spaces*, Journal of Machine Learning Research **13** (2012), no. Nov, 3247–3251.

[110] N Rosenblum, X Zhu, and B Miller, *Who wrote this code? Identifying the authors of program binaries*, Computer SecurityESORICS 2011 (2011).

[111] Nathan Rosenblum, BP Miller, and Xiaojin Zhu, *Recovering the toolchain provenance of binary code*, Proceedings of the 2011 International ... (2011), 100–110.

[112] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu, *Extracting compiler provenance from program binaries*, Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10 (2010), 21.

[113] J. Sahs and L. Khan, *A machine learning approach to android malware detection*, Intelligence and Security Informatics Conference (EISIC), 2012 European, Aug 2012, pp. 141–147.

[114] P. Sallis, a. Aakjaer, and S. MacDonell, *Software forensics: old methods for a new science*, Proceedings 1996 International Conference Software Engineering: Education and Practice (1996), 481–485.

[115] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy, *Android permissions: A perspective combining risks and benefits*, Proceedings of the 17th ACM SACMAT '12 (New York, NY, USA), ACM, 2012, pp. 13–22.

[116] Daniel Schreckling, Joachim Posegga, Johannes Köstler, and Matthias Schaff, *Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android*, Proceedings of the 6th WISTP'12 (Berlin, Heidelberg), Springer-Verlag, 2012, pp. 208–223.

[117] Intel Security, *Mobile threat report: What's on the horizon for 2016*, White Paper, January 2016.

[118] James Sellwood and Jason Crampton, *Sleeping android: The danger of dormant permissions*, Proceedings of the 3rd SPSM '13 (New York, NY, USA), ACM, 2013, pp. 55–66.

[119] A. Shabtai, Y. Fledel, and Y. Elovici, *Automated static code analysis for classifying android applications using machine learning*, Computational Intelligence and Security (CIS), 2010 International Conference on, Dec 2010, pp. 329–333.

[120] V.Y. Shen, S.D. Conte, and H.E. Dunsmore, *Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support*, IEEE Transactions on Software Engineering **SE-9** (1983), no. 2, 155–165.

[121] Eugene H Spafford, *The internet worm program: An analysis*, ACM SIGCOMM Computer Communication Review **19** (1989), no. 1, 17–57.

[122] Eugene H. Spafford and Stephen A. Weeber, *Software Forensics: Can We Track Code to its Authors?*, 1992.

[123] Michael Spreitzenbarth, *Current android malware*, `https://forensics.spreitzenbarth.de/android-malware/`, [Online; accessed Oct 14, 2016].

[124] Benno Stein, Nedim Lipka, and Peter Prettenhofer, *Intrinsic plagiarism analysis*, Lang. Resour. Eval. **45** (2011), no. 1, 63–82.

[125] Android Team, *Building process for an app.*, https://developer.android.com/studio/build/index.html, [Online; accessed April 27, 2014].

[126] F-Droid Team, *F-droid*, http://f-droid.org.

[127] KL Verco and MJ Wise, *Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems*, ACSE (1996).

[128] N. Viennot, E. Garcia, and J. Nieh, *A measurement study of Google Play*, Proceedings of SIGMETRICS 2014, 2014, p. 13.

[129] Andrew Walenstein and Arun Lakhotia, *The software similarity problem in malware analysis*, ..., and Similarity in Software (2007), 1–10.

[130] Cheng Wang, Jianmin Pang, Rongcai Zhao, Wen Fu, and Xiaoxian Liu, *Malware detection based on suspicious behavior identification*, Education Technology and Computer Science, 2009. ETCS'09. First International Workshop on, vol. 2, IEEE, 2009, pp. 198–202.

[131] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu, *Behavior based software theft detection*, Proceedings of the 16th ACM conference on Computer and communications security - CCS '09 (2009), 280.

[132] Carsten Willems, Thorsten Holz, and Felix Freiling, *Toward automated dynamic malware analysis using cwsandbox*, IEEE Security and Privacy **5** (2007), no. 2, 32–39.

[133] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu, *Droidmat: Android malware detection through manifest and api calls tracing*, Proceedings of the 7th Asia JCIS, Aug 2012, pp. 62–69.

[134] Yoon-chan Jhi Xinran, Wang Xiaoqi, Jia Sencun, Zhu Peng, and Liu Dinghao, *Value-Based Program Characterization and Its Application to Software Plagiarism Detection*, ICSE 11, 2011, pp. 756–765.

[135] Lok Kwong Yan and Heng Yin, *Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis*, Proceedings of the 21st USENIX Conference on Security Symposium (Berkeley, CA, USA), Security'12, USENIX Association, 2012, pp. 29–29.

[136] Yandex, *Yandex store*, https://store.yandex.com/.

[137] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu, *Appspear: Bytecode decrypting and dex*

reassembling for packed android malware, pp. 359–381, Springer International Publishing, Cham, 2015.

[138] Rowland Yu, *Android packers: facing the challenges, building solutions*, Proceedings of the Virus Bulletin Conference (VB'14), 2014, pp. 266–275.

[139] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu, *Viewdroid: Towards obfuscation-resilient mobile application repackaging detection*, 7th ACM Conference on WiSec 2014 (Oxford, United Kingdom), 2014.

[140] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang, *Vetting undesirable behaviors in android apps with permission use analysis*, Proceedings of the 2013 ACM SIGSAC (New York, NY, USA), CCS '13, ACM, 2013, pp. 611–622.

[141] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou, *Fast, scalable detection of "piggybacked" mobile applications*, Proceedings of the CODASPY '13 (New York, NY, USA), ACM, 2013, pp. 185–196.

[142] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning, *Detecting repackaged smartphone applications in third-party android marketplaces*, Pro-

ceedings of the Second ACM CODASPY '12 (New York, NY, USA), ACM, 2012, pp. 317–326.

[143] Yajin Zhou and Xuxian Jiang, *Dissecting Android malware: Characterization and evolution*, IEEE Symposium on Security and Privacy (SP), IEEE, 2012, pp. 95–109.

[144] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang, *Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets*, 19th Annual NDSS, 2012.

Appendix A

Evolution of studies about Authorship Attribution

A.1 Bibliography

Here we present bibliography that represent the evolution about Authorship Attribution refered in Figure 2.5.

1. Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. *cis.syr.edu*, 2013.

2. A.J. Albrecht and J. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, Nov. 1983.

3. D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

4. R. Bayer, M. Halstead, and N. Bulut. Experimental validation of a struc-

tural property of fortran algorithms. *ACM '74 Proceedings of the 1974 annual conference*, pages 207–211, 1974.

5. H. Berghel and D. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, (August), 1984.

6. P. V. Bhattathiripad. Software Piracy Forensics: A Proposal for Incorporating Dead Codes and Other Programming Blunders as Important Evidence in AFC Test. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 206–212, July 2012.

7. N. Bulut and M. Halstead. Invariant properties of algorithms. *ACM SIGPLAN Notices*, pages 12–13, 1973.

8. S. Burrows and S. M. M. Tahaghoghi. Source Code Authorship Attribution using n -grams. In *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*, pages 32–39, 2007.

9. S. Burrows, A. L. Uitdenbogerd, and A. Turpin. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience*, 44(August 2012):1–32, 2014.

10. D.-K. Chae, S.-W. Kim, J. Ha, S.-C. Lee, and G. Woo. Software plagiarism detection via the static API call frequency birthmark. *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*, page 1639, 2013.

11. S. Choi, H. Park, H.-i. Lim, and T. Han. A static API birthmark for Windows binary executables. *Journal of Systems and Software*, 82(5):862–873, May 2009.

12. R. Chouchane, N. Stakhanova, A. Walenstein, and A. Lakhotia. Detecting machine-morphed malware variants via engine attribution. *Journal of Computer Virology and Hacking Techniques*, 2013.

13. CrowdStrike. Putter panda. Intelligence Report, June 2014.

14. B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering*, SE-5(2):96–104, Mar. 1979.

15. C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead mostly static JavaScript malware detection. *Proceedings of the Usenix Security Symposium*, 2011.

16. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

17. H. Ding and M. H. Samadzadeh. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software*, 72(1):49–57, June 2004.

18. J. Donaldson, A. Lancaster, and P. Sposato. A plagiarism detection system. *ACM SIGCSE Bulletin*, pages 21–25, 1981.

19. J. Faidhi and S. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, 1987.

20. A. Fitzsimmons and T. Love. A review and evaluation of software science. *ACM Computing Surveys (CSUR)*, 10(1), 1978.

21. J. Forristal. Android: One root to own them all. In *BlackHat*, 2013.

22. G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software*, 81(3):447–460, Mar. 2008.

23. G. Frantzeskou and E. Stamatatos. Source code author identification based on n-gram author profiles. *Artificial Intelligence Applications and Innovations*, 2006.

24. G. Frantzeskou and E. Stamatatos. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence*, 6(1):1–18, 2007.

25. G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas. Effective identification of source code authors using byte-level information. *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 893, 2006.

26. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

27. H. Gonzalez, N. Stakhanova, and A. Ghorbani. Droidkin: Lightweight detection of android apps similarity. In *Proceedings of International Conference on Security and Privacy in Communication Networks (SecureComm 2014)*, 2014.

28. M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *The 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 281–294, 2012.

29. S. Grier. A tool that detects plagiarism in pascal programs. In *ACM SIGCSE Bulletin*, volume 13, pages 15–20. ACM, 1981.

30. M. Halstead. Toward a theoretical basis for estimating programming effort. *Proceedings of the 1975 annual conference*, pages 222–224, 1975.

31. M. H. Halstead. Natural laws controlling algorithm structure? *ACM Sigplan Notices*, pages 19–26, 1972.

32. P. Hamer and G. Frewin. MH Halstead's Software Science-a critical examination. *Proceedings of the 6th international conference on Software engineering*, pages 197–206, 1982.

33. J. Hayes. Authorship Attribution: A Principal Component and Linear Discriminant Analysis of the Consistent Programmer Hypothesis. *IJ Comput. Appl.*, 2008.

34. J. H. Hayes and J. Offutt. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability*, 20(4):329–356, Dec. 2010.

35. H. T. Jankowitz. Detecting plagiarism in student pascale programs. *The Computer Journal*, 31(1):1–8, 1988.

36. J.-H. Ji, G. Woo, and H.-G. Cho. A Plagiarism Detection Technique for Java Program Using Bytecode Analysis. *2008 Third International Conference on Convergence and Hybrid Information Technology*, pages 1092–1098, Nov. 2008.

37. I. Krsul. Authorship Analysis: Identifying the Author of a Program. Technical report, Purdue University, 1994.

38. I. Krsul and E. Spafford. Authorship analysis: Identifying the author of a program. 1995.

39. I. Krsul and E. H. Spafford. Authorship analysis: identifying the author of a program. *Computers & Security*, 16(3):233–257, Jan. 1997.

40. K. Lab. Global corporate it security risks: 2013. White Paper, May 2013.

41. R. C. Lange and S. Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*, page 2082, 2007.

42. R. Layton, P. Watters, and R. Dazeley. Automatically determining phishing campaigns using the USCAP methodology. *2010 eCrime Researchers Summit*, pages 1–8, 2010.

43. Mandiant. Apt1: Exposing one of chinas cyber espionage units. White Paper, Feb. 2013.

44. P. Oman and C. Cook. Programming style authorship analysis. *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 320–326, 1989.

45. K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin*, pages 30–41, 1976.

46. a. Parker and J. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, May 1989.

47. L. Prechelt, G. Malpohl, and M. Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *J. UCS*, 8(11):1016–1038, 2002.

48. P. Rascagneres. Apt1: technical backstage. White Paper, Mar. 2013.

49. D. K. S. Reddy and A. K. Pujari. N-gram analysis for computer virus detection. *Journal in Computer Virology*, 2(3):231–239, Nov. 2006.

50. N. Rosenblum, B. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. pages 100–110, 2011.

51. N. Rosenblum, X. Zhu, and B. Miller. Who wrote this code? Identifying the authors of program binaries. *Computer SecurityESORICS 2011*, 2011.

52. N. E. Rosenblum. *The Provenance Hierarchy of Computer Programs*. PhD thesis, WISCONSINMADISON, 2011.

53. N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10*, page 21, 2010.

54. P. Sallis, a. Aakjaer, and S. MacDonell. Software forensics: old methods for a new science. *Proceedings 1996 International Conference Software Engineering: Education and Practice*, pages 481–485, 1996.

55. V. Shen, S. Conte, and H. Dunsmore. Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *IEEE Transactions on Software Engineering*, SE-9(2):155–165, Mar. 1983.

56. E. H. Spafford. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

57. E. H. Spafford and S. A. Weeber. Software Forensics: Can We Track Code to its Authors?, 1992.

58. K. Verco and M. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *ACSE*, 1996.

59. A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. pages 1–10, 2007.

60. X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*, page 280, 2009.

61. Y.-c. J. Xinran, W. Xiaoqi, J. Sencun, Z. Peng, and L. Dinghao. Value-Based Program Characterization and Its Application to Software Plagiarism Detection. In *ICSE 11*, pages 756–765, 2011.

62. L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

63. F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *7th ACM Conference on WiSec 2014*, Oxford, United Kingdom, 2014.

64. M. Zheng, M. Sun, and J. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.

65. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

66. Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109. IEEE, 2012.

67. Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

Vita

Candidate's full name: Hugo F. Gonzalez Robledo
M.Sc. in Computer Science, Technological Institute of San Luis Potosi, Mexico. 2005.
Bachelor of Software Engineering, Technological Institute of San Luis Potosi, Mexico, 2000.
Publications:

- Gonzalez, H., Stakhanova, N., & Ghorbani, A. A. (2014, September). Droidkin: Lightweight detection of android apps similarity. In International Conference on Security and Privacy in Communication Systems (pp. 436-453). Springer International Publishing.
- Gonzalez, H., Kadir, A. A., Stakhanova, N., Alzahrani, A. J., & Ghorbani, A. A. (2015, April). Exploring reverse engineering symptoms in Android apps. In Proceedings of the Eighth European Workshop on System Security (p. 7). ACM.
- Jain, A., Gonzalez, H., & Stakhanova, N. (2015, December). Enriching reverse engineering through visual exploration of Android binaries. In Proceedings of the 5th Program Protection and Reverse Engineering Workshop (p. 9). ACM.
- Gonzalez, H., Stakhanova, N., & Ghorbani, A. A. (2016, December). Measuring code reuse in Android apps. In Privacy, Security and Trust (PST), 2016 14th Annual Conference on (pp. 187-195). IEEE.

Online academic profiles:

- Google. https://scholar.google.ca/citations?user=yduXQc4AAAAJ
- Researchgate. https://www.researchgate.net/profile/Hugo_Gonzalez