Similarity Based Large Scale Malware Analysis: Techniques and Implications

by

Yuping Li

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Xinming Ou, Ph.D.
Lawrence Hall, Ph.D.
Jarred Ligatti, Ph.D.
Nasir Ghani, Ph.D.
Jiyong Jang, Ph.D.

Date of Approval:
June 13, 2018

Keywords: Binary Similarity Analysis, Malware Clustering, Malware Detection, Machine Learning

Copyright © 2018, Yuping Li
DEDICATION

To my family
ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to my advisor Xinming Ou, for his continuous support and help during my Ph.D. life. He gave me tremendous insightful guidance and invaluable advice for critical thinking, and provided me great opportunities to conduct meaningful research for solving real-world problems. I feel truly thankful to be his student and work with him over the past several years.

I feel truly indebted to Dr. Lawrence Hall, Dr. Jarred Ligatti, Dr. Nasir Ghani and Dr. Jiyong Jang for serving on my dissertation committee. This dissertation would not be possible without their numerous constructive feedback and extremely helpful suggestions. In particular, Dr. Jang is always an amazing collaborator and his early work on software security analysis has been an incredible source of inspiration; Dr. Hall’s commitment to excellence greatly improved my paper writing and his broad knowledge consistently helped me to conduct the Machine Learning related research in the right way.

I truly appreciate all the talented coauthors and colleagues for their help and critical feedback for all the great work. Special thanks go to all the past and current members of Argus Cyber Security Lab for their wonderful friendship that makes my Ph.D. life more exciting and memorable.

This work was partially supported by the U.S. National Science Foundation under Grant No. 1314925 and 1622402.
# TABLE OF CONTENTS

LIST OF TABLES iv  
LIST OF FIGURES vi  
ABSTRACT vii  

## CHAPTER 1: INTRODUCTION  
1.1 Background 1  
1.2 Malware Analysis 2  
1.2.1 Desktop Malware Analysis 3  
1.2.2 Android Malware Analysis 5  
1.2.3 Machine Learning Based Malware Analysis 7  
1.3 Objectives 9  
1.4 Contributions 11  
1.4.1 Accurate and Efficient Binary Similarity Analysis Tool 11  
1.4.2 Specialized Android Malware Clustering System 11  
1.4.3 Practical Implications of ML Based Android Malware Detection 12  
1.5 Organization of the Dissertation 12  

## CHAPTER 2: ACCURATE AND EFFICIENT BINARY SIMILARITY ANALYSIS  
2.1 Approach Overview 16  
2.1.1 Basic Block Type Abstraction 16  
2.1.2 Blended n-gram Graphical Features 18  
2.1.3 Graph Signature Format and Comparison 19  
2.1.4 Fuzzy Hash Generation and Comparison 21  
2.2 Evaluation of CFG Similarity Analysis Algorithms 22  
2.2.1 Existing CFG Similarity Comparison Algorithms 23  
2.2.2 CFG Comparison Algorithm Evaluation Strategy 24  
2.2.3 Experiment Data Preparation 26  
2.2.4 Evaluation Results 27  
2.2.5 Overall Performance 30  
2.3 Evaluation of Binary Similarity Analysis Tools 31  
2.3.1 Binary Graph Signature Accuracy 32  
2.3.2 Graphical Fuzzy Hash Accuracy 33  
2.3.3 Similarity Tool Output Consistency 34  
2.3.4 Similarity Tool Accuracy Analysis 35  
2.3.5 Malware Detection Analysis 37  
2.3.6 Obfuscation State Analysis 39
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Combined basic block types</td>
<td>18</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Optimal clustering results for different algorithms</td>
<td>27</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Distance ranges for different CFG pairs</td>
<td>29</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>Time taken to finish all pairwise distance calculation</td>
<td>31</td>
</tr>
<tr>
<td>Table 2.5</td>
<td>Malicious ground truth dataset</td>
<td>32</td>
</tr>
<tr>
<td>Table 2.6</td>
<td>BINGSim-O Accuracy with different n-gram sizes</td>
<td>33</td>
</tr>
<tr>
<td>Table 2.7</td>
<td>Optimal clustering results for different tools</td>
<td>36</td>
</tr>
<tr>
<td>Table 2.8</td>
<td>Benign ground truth dataset</td>
<td>38</td>
</tr>
<tr>
<td>Table 2.9</td>
<td>Malware detection results for similarity analysis tools</td>
<td>38</td>
</tr>
<tr>
<td>Table 2.10</td>
<td>Packer identification results</td>
<td>40</td>
</tr>
<tr>
<td>Table 2.11</td>
<td>Similarity calculation impacts for similarity analysis tools</td>
<td>40</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Major feature categories and main differences comparing with Juxtapp</td>
<td>50</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Clearly labeled malware families</td>
<td>60</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Clustering results of different sub-version datasets</td>
<td>63</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Malicious payload under popular libraries</td>
<td>66</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Summary of the prepared datasets</td>
<td>78</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Experiment results for B-2016 dataset using malware prediction techniques</td>
<td>80</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Experiment results for U dataset using malware prediction techniques</td>
<td>80</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Experiment results for B-2016 dataset using mislabel identification techniques</td>
<td>82</td>
</tr>
</tbody>
</table>
Table 4.5  Experiment results for U dataset using mislabel identification techniques  

Table 4.6  Manually confirmed zero-day malware samples
LIST OF FIGURES

Figure 2.1 The BingSim workflow 16
Figure 2.2 Distribution of in-degree value, out-degree value and combined basic block types 17
Figure 2.3 A simple CFG and its blended N-gram features 18
Figure 2.4 Single-linkage clustering results for different CFG comparison algorithms 28
Figure 2.5 Average-linkage clustering results for different CFG comparison algorithms 29
Figure 2.6 Complete-linkage clustering results for different CFG comparison algorithms 30
Figure 2.7 Cumulative distance distribution for different CFG comparison algorithms 31
Figure 2.8 Mean and Std. 34
Figure 2.9 Histogram 34
Figure 2.10 Single-linkage clustering results for different tools 37
Figure 3.1 Overview of the clustering system with five Android malware samples. 47
Figure 3.2 Overall fingerprint generation procedure 49
Figure 3.3 Example procedure to safely remove legitimate “twitter4j” library code 51
Figure 3.4 Extracting a candidate payload from two malware applications 52
Figure 3.5 Random feature collision status 61
Figure 3.6 Benign apps lib removal accuracy 61
Figure 3.7 Clustering analysis results for Android genome dataset 63
Figure 4.1 Malware prediction experiment overview 72
Figure 4.2 Mislabeled identification experiment overview 74
ABSTRACT

Malware analysis and detection continues to be one of the central battlefields for cybersecurity industry. For the desktop malware domain, we observed multiple significant ransomware attacks in the past several years, e.g., it was estimated that in 2017 the WannaCry ransomware attack affected more than 200,000 computers across 150 countries with hundreds of millions damages. Similarly, we witnessed the increased impacts of Android malware on global individuals due to the popular smartphone and IoT devices worldwide. In this dissertation, we describe similarity comparison based novel techniques that can be applied to achieve large scale desktop and Android malware analysis, and the practical implications of machine learning based approaches for malware detection.

First, we propose a generic and effective solution for accurate and efficient binary similarity analysis of desktop malware. Binary similarity analysis is an essential technique for a variety of security analysis tasks, including malware detection and malware clustering. Even though various solutions have been developed, existing binary similarity analysis methods still suffer from limited efficiency, accuracy, and usability. In this work, we propose a novel graphical fuzzy hashing scheme for accurate and efficient binary similarity analysis. We first abstract control flow graphs (CFGs) of binary codes to extract blended n-gram graphical features of the CFGs, and then encode the graphical features into numeric vectors (called graph signatures) to measure similarity by comparing the graph signatures. We further leverage a fuzzy hashing technique to convert the numeric graph signatures into smaller fixed size fuzzy hash outputs for efficient comparisons. Our comprehensive evaluation demonstrates that our blended n-gram graphical feature based CFG comparison is more effective and efficient compared to existing CFG comparison techniques. Based on our CFG comparison method, we develop BINGSim, a binary similarity analysis tool, and show that BINGSim outperforms existing binary similarity analysis tools while conducting similarity analysis based malware detection and malware clustering.
Second, we identify the challenges faced by overall similarity based Android malware clustering and design a specialized system for solving the problems. Clustering has been well studied for desktop malware analysis as an effective triage method. Conventional similarity-based clustering techniques, however, cannot be immediately applied to Android malware analysis due to the excessive use of third-party libraries in Android application development and the widespread use of repackaging in malware development. We design and implement an Android malware clustering system through iterative mining of malicious payloads and checking whether malware samples share the same version of malicious payloads. Our system utilizes a hierarchical clustering technique and an efficient bit-vector format to represent Android apps. Experimental results demonstrate that our clustering approach achieves precision of 0.90 and recall of 0.75 for the Android Genome malware dataset, and average precision of 0.98 and recall of 0.96 with respect to manually verified ground-truth.

Third, we study the fundamental issues faced by traditional machine learning (ML) based Android malware detection systems, and examine the role of ML for Android malware detection in practice, which leads to a revised evaluation strategy that evaluates an ML based malware detection system by checking their zero-day detection capabilities. Existing machine learning based Android malware research obtains the ground truth by consulting AV products, and uses the same label set for training and testing. However, there is a mismatch between how the ML system has been evaluated, and the true purpose of using ML system in practice. The goal of applying ML is not to reproduce or verify the same potentially imperfect knowledge, but rather to produce something that is better — closer to the ultimate ground truth about the apps’ maliciousness. Therefore, it will be more meaningful to check their zero-day detection capabilities than detection accuracy for known malware. This evaluation strategy is aligned with how an ML algorithm can potentially benefit malware detection in practice, by acknowledging that any ML classifier has to be trained on imperfect knowledge, and such knowledge evolves over time. Besides the traditional malware prediction approaches, we also examine the mislabel identification approaches. Through extensive experiments, we demonstrate that: (a) it is feasible to evaluate ML based Android malware detection systems with regard to their zero-day malware detection capabilities;
(b) both malware prediction and mislabel identification approaches can be used to achieve verifiable zero-day malware detection, even when trained with an old and noisy ground truth dataset.
CHAPTER 1: INTRODUCTION

1.1 Background

Malicious software, a.k.a. malware, is an generic term used to refer to a variety of intentionally hostile, harmful, or intrusive software programs, such as virus, worms, Trojan, bot, backdoor, etc. In this work, we mainly consider the malicious programs for the Windows platform, and the malicious applications from the Android platform at the same time, even though they need separated detection and analysis techniques.

Dating back 2014, executives from the leading antivirus (AV) company Symantec already claimed [1] that antivirus was “dead” because: (1) companies were gradually shifting from protect to detect and response; and (2) dedicated hackers can get into a system even if AV products were installed. However, we observe that: (a) more companies were building AV detection solutions in recent years, e.g., the number of AV products listed on VirusTotal [2] increased from around 40 in 2010 to about 70 in 2018; (b) all size organizations and companies are still heavily using the AV products as an important layer of security defense mechanism.

Indeed, malware remains one of the major attacking tools for cyber criminals and is responsible for many notorious cyber attacks in recent years. Besides spying on user activity and harvesting for credentials, malware evolved with more advanced approaches for amplifying the damage and gaining profits. For example, Mirai [3] was a malware that continuously scans for networked Internet of Things (IoT) devices and turns the IoT devices into remotely controlled “bots”. The Mirai botnet was used to launch multiple large-scale distributed denial of service attacks (DDoS) in 2016; Utilizing the NSA leaked EternalBlue and DoublePulsar exploits, hackers created WannaCry [4] ransomware which infected more than 200,000 computers across 150 countries in 2017; Thanks to the boom of cryptocurrencies in 2017, miscreants also started to create cryptomining
malware (e.g., Trickbot) which turns infected machines into coin miners thereby stealthily collecting lucrative cryptocurrency profits.

Due to the increased popularity of smartphone devices, we also noticed the gradually increasing malware threats in recent years on mobile platforms, especially for Android. Like all the trends that we have observed on the Windows platform, hackers also developed various creative techniques for Android malware to spy on user activity, steal personal or financial data, clear suspicious activity traces and maintain persistence access, escalate to administrator privilege using root exploits, or multiple ways to achieve anti-analysis using string renaming, encryption, dynamic loading, native payload, etc. Over the years, cyber criminals developed different approaches for monetization, such as sending premium SMS messages, selling the stolen banking credentials, showing aggressive advertisements, and carrying out fraud campaigns. In addition, the emerging threats observed from the desktop malware domain were also found within Android malware domain. For instance, in 2017 researchers [5] found a type of ransomware named Double Locker which can change the PIN number and encrypt the data on Android devices.

Comparing to malware threats in the past, system vulnerability (e.g., exploiting) and human vulnerability (e.g., phishing) remains the main vehicles for spreading the malware, but nowadays malware authors have more options to profit massively within a short period of time, and a successful malware campaign typically causes more severe financial damages to impacted organizations or individuals. Malware analysis is an on-going arm race process between security defenders and cyber criminals, we believe malware will remain to be one of the major security threats in future and malware analysis will continue to be a necessary and relevant technique within security industry.

1.2 Malware Analysis

In the real world, malware authors can reuse existing code to create new instances of malware that have the same functionality as existing malware, those newly created malware instances are often referred as malware “variants”. Malware variants are observed within both the desktop malware and Android malware domain, thus a single malware family may contain a huge number of different malware samples, and many of them share the same malware functionalities.
To cope with the ever increasing number of real world malware samples that are variants of existing malware families, similarity analysis techniques were frequently used for creating scalable malware analysis solutions. For example, analyzing certain samples of the malware family thoroughly, then extracting the critical signatures for that family and checking whether other samples also present the same signatures. Or converting all samples into an efficient feature representation, then checking whether the newly processed samples are similar to malware samples in their feature representation.

Over the past two decades, lots of researchers have worked in the malware analysis domain and provided numerous techniques or systems for handling specific malware analysis tasks. In this work, we mainly aim to categorize previous research works that are related to similarity based malware analysis, and do not intend to cover all previous research works in the malware analysis domain. We separate similarity based malware analysis into the following categories: desktop malware analysis, Android malware analysis, and machine learning based malware analysis.

1.2.1 Desktop Malware Analysis

The early works in the desktop malware analysis domain were pioneered by anti-virus companies, which aimed to develop standalone detection systems by providing distinctive malware signatures for each malware family. In the early days, malware signatures were often carefully created by thorough manual analysis to avoid false positives, which is labor intensive and time consuming. Hancock [6] was the first string signature generation system that automatically generated contiguous byte sequences as signatures, it employed various categories of heuristics to create string signatures with a sufficiently low false positive rate. Tabish [7] designed a malware detection system using statistical analysis of byte-level file content, it achieved 90% malware detection without relying on specific byte sequences or strings of the file content. Jang [8] provided a malware clustering system called BitShred by extracting n-gram features from malware code section byte sequences and applying feature hashing technique to dramatically reduce the feature space. BitShred was experimentally shown to be effective and scalable for malware clustering. Jacob et al. [9] presented packer-agnostic filter approach to detect similar malware samples by firstly categorizing malware code into unpacked, compressed, encrypted, and multi-layer encrypted, then applying different sim-
ilarity measurements accordingly. ssdeep [10] was a representative fuzzy hashing algorithm that was used to detect homologous files using context triggered piecewise hashes, it couldn’t recognize significant number of samples that are known to be similar due to the byte sequences level operations adopted. nextGen-hash [11] was a concretized fuzzy hashing approach based on the main ideas developed in BitShred, which generated signatures from the program code section, but its significant fingerprint size made it hard to use in practice.

In addition to byte sequences malware analysis, researchers also explored malware detection through analysis of instruction sequences (e.g., assembly opcodes), which provides richer semantic information compared to low level byte sequences. Christodorescu et al. [12] presented a malware detection algorithm by incorporating instruction semantics to detect malicious program traits, and showed that it can detection malware variants with a relatively low run-time overhead. Santos [13] provided a malware detection solution based on the frequency of appearance of opcode sequences and described an method to mine the relevance of opcodes and weight the opcode sequence frequency. Moskovitch [14] both proposed n-gram features from opcode sequences, and demonstrated it to be more effective than n-gram features extracted from byte sequences. MutantX-S [15] was a scalable framework designed to effectively cluster a large number of malware samples by extracting n-gram features from disassembled program opcode sequences, and showed that the system can achieve a good balance between accuracy and scalability.

Instruction sequences provide more semantics information than byte sequences, but they are still susceptible to obfuscation and suffer from accuracy in reality. Therefore, analyzing malware based on the graph representation of malware samples was also adopted in the security community. Bruschi [16] proposed to detect the metamorphic malicious code inside a program based on the comparison of the CFGs of the program against the CFGs of known malware, and provided experimental data supporting the validity of the strategy. Hu [17] designed a multi-resolution malware indexing system that can efficiently detect malware by comparing the similarity of function-call graphs and demonstrated effective detection capabilities. Cesare et al. [18] supplied a similarity search approach using distance metrics of malware control flow graphs (CFG). They created malware signatures for the malware by decomposing the CFGs into either fixed size k-subgraphs (from
structural representation), or q-gram strings (from string representation). Kong [19] presented a generic framework that extracts structural information from malware programs as attributed function call graphs, and learns discriminant malware distance metrics and evaluate the similarity between the attributed function call graphs of two malware programs. Similarly, Alam [20] provided a malware detection scheme using Annotated Control Flow Graph (ACFG), and demonstrated that ACFG generated more accurate results than CFG based detection approach.

The majority of existing similarity based desktop malware analysis works are based on static analysis. A higher level of program representation was shown to be more resilient against instruction level obfuscation, but still suffered from advanced obfuscation such as compressing, encryption, and virtualization etc. In order to fight against malware obfuscation, dynamic analysis based systems were also developed, the dynamic behaviors or traces collected from such systems were typically further forwarded for anomaly analysis or machine learning based analysis, we therefore describe such systems in Section 1.2.3.

1.2.2 Android Malware Analysis

An Android app is an archive file with a “apk” extension, it commonly contains program’s code (“.dex” files), resources, assets, certificates, and manifest file. Android apps are typically developed in the Java language and compiled to bytecode, then converted from Java Virtual Machine compatible “.class” files to Dalvik virtual machine compatible “.dex” files before installation on Android-powered devices. The Android operating system is a multi-user Linux system in which each app is a different user and lives in its own security sandbox.

Being a Java application, it becomes easier for attackers to disassemble the compiled app code into an intermediate representation (e.g., smali) or even source code representation, which subsequently make it easier to create new Android apps by repackaging existing legitimate and popular Android apps. This malicious practice was frequently used used by miscreants to replace in-app advertisements or reroute revenues or inject backdoor or malicious payloads within the repackaged apps.

A significant amount of early research in this domain was mainly designed for detection of repackaged malware, and similarity analysis between malicious apps and known benign apps was
adopted because of the nature of the app repackaging procedure. Zhou et al. [21] developed Droid-MOSS to effectively localize and detect the changes from app repackaging behavior by using a fuzzy hashing technique to disassembled instruction sequences. Hanna et al. designed Juxtapp [22], a scalable infrastructure for code similarity analysis among Android applications by extracting n-gram features from instruction sequences and applying a feature hashing technique. DNADroid [23] can be used to analyze Android programs and compare program dependency graphs between methods in candidate applications. In addition, Lin [24] suggested to identify repackaged apps by checking thread-grained system call sequences, which does not require the original benign apps to identify the repackaged apps. Besides analysis from Dalvik instruction sequences, core application resources were also used for detecting repackaged apps [25], because of the close relationships between the original app and the repackaged app. It can identify repackaged apps efficiently and effectively even if the code has been obfuscated. Zhang proposed ViewDroid [26], a user interface based repackaging detection system through view graphs of the app. View graph represents a new birthmark of the app that can characterize Android apps from a higher level abstraction, which also make it resilient to code obfuscation.

Even though clustering analysis systems were shown to be effective for large scale desktop malware triage, the traditional similarity based clustering systems cannot be immediately applied for Android malware clustering because of the following reasons: (1) the malicious code segments often constitute only a small fraction of an Android malware sample. (2) there are excessive usages of third-party legitimate libraries in Android application development, which means samples from different families may share high level of overall similarity when they both contain the same large library code; (3) attackers frequently use repackaging techniques to create new malware samples, which means samples from the same families may share a low level of overall similarity when they are repackaged based on different apps. Over the past few years, we only observed a few systems that tried to address the Android malware clustering problem, but the overall performance was limited because of the above constraints. For example, Samra [27] extracted features from Android app manifest files, and could only cluster applications into two categories using the K-means algorithm. ClusTheDroid [28] was a system for clustering Android malware using 38 features extracted from
profiles of reconstructed dynamic behaviors. Without properly excluding the features or behaviors that belong to the original benign apps or legitimate libraries, traditional clustering approaches would not be able to produce promising results.

As shown above, similarity analysis techniques in the Android domain were mainly used to identify repackaged malicious apps by comparing them against known legitimate apps instead of comparing them against known malicious apps. Repackaged malware represents a significant portion of Android malware, but it has been observed [29] that standalone malware is increasing in recent years. In order to detect malware based on knowledge of existing malware samples, malicious payload identification and extraction is essential. Zhou and Jiang [30] manually analyzed malicious payloads of Android malware and summarized the findings in the Android Malware Genome project. DroidAnalytics [31] presented a multi-level signature based analytics system to examine and associate repackaged Android malware. AndroSimilar [32] was a signature based malware detection solution utilizing statistically improbable features, it applied a syntactic foot-printing mechanism to detect unknown malware by identifying statistically similar regions. MassVet [33] analyzed graph similarity at the function level and extracted the shared non-legitimate functions as malicious payloads through commonality and differential analysis, and it applied a whitelist to exclude legitimate library code from analysis. MassVet was shown to be effective for vetting Android apps, but it failed to extract the malicious payloads if they are injected under popular library namespaces or under existing functions.

1.2.3 Machine Learning Based Malware Analysis

Machine learning (ML) techniques have been applied for both desktop malware and Android malware analysis. Overall, ML based malware detection first converts malware samples into feature vectors through either static analysis or dynamic analysis, then trains on a labeled ground truth dataset and creates ML models for prediction. Essentially, the overall process tries to abstract certain implicit connections from the training dataset, and checks whether the testing samples are “similar” to the training samples in the features according to ML algorithms applied. In this section, we mainly discuss ML based malware detection systems that are commonly referred as malware classification or malware prediction.
ML based desktop malware detection systems using static features dates back 2001. Schultz et al. [34] were among the first to suggest using data mining methods for detection of new malicious executables. They explored various machine learning algorithms with different static features for malware classification and claimed that the ML detection rate was twice as compared to signature based method. Kolter [35] later tried n-gram features with different classifiers and concluded that the boosted decision tree gave the best classification results. In [36], Tian et al. attempted to classify Trojans with the frequency of function length, which was measured by the number of bytes in the code. Siddiqui [37] used variable length instruction sequence along with machine learning for detecting worms in the wild. Santos [38] designed a semi-supervised learning approach which was able to learn from labeled and unlabeled data and provides a solution with respect to the intrinsic structure displayed by both labeled and unlabeled instances. Nataraj [39] proposed a method for visualizing and classifying malware using image processing techniques, which visualize malware binaries as gray-scale images. the authors compared binary texture based analysis (based on image processing techniques) with that of dynamic analysis.

Later, researchers began to apply dynamic techniques to improve the accuracy and effectiveness of malware classification systems. In [40, 41], Rieck et al. suggested to classify malware variants using their shared behavioral patterns. They used existing anti-virus products to prepare labeled malware, then monitored the behavior of malware samples in a sandbox environment. In order to apply the learning algorithms, they provided a way to effectively embed the observed behavior in a vector space. Bayer [42] designed a system that clusters large sets of malicious binaries based on their behavior reports. Their behavior reports are collected from Anubis sandbox and the approximate clustering algorithm was based on Locality Sensitive Hashing. Zolkipli [43] presented a malware classification approach using malware behavior reports collected CWSandbox and Anubis sandbox environments, but they relied on human analysis for processing the dynamic behavior reports.

In recent years, various ML based systems were also proposed for Android malware detection. Sahs [44] presented a ML based which uses static features such as permissions and trains a One-class SVM in an offline manner. Their experiments showed a low false negative rate but high false
positive rate. DroidMat [45] used a static analyst paradigm for detecting the Android malware, it utilized static information including permissions, deployment of components, Android Intent information and API calls for characterizing the Android applications behavior. Peiravian [46] proposed to combine permission and API (Application Program Interface) calls and use machine learning methods to detect malicious Android Apps, and validates the algorithm performance using real world apps. MAST [47] helped resource intensive operations (e.g., manual analysis) to triage their priority, thereby reducing the average computation overhead. This system utilizes a statistical method called Multiple Correspondence Analysis (MCA), and uses permissions, intents and the presence of native code to determine the probabilities of being malicious. Drebin [48] gathered a massive set (more than 500K) of features which contain different types of manifest features (permissions, etc.) and ‘code’ features (URLs, APIs etc.). It applied a SVM to train a detection model which later can be uploaded to a device to do on-device malware detection, and generated impressive performance results. DroidMiner [49] was a malicious Android app detection system that uses static analysis to automatically mine malicious program logic from known Android malware by computing the alignment of an application’s behavioral graph to a set of known malicious modalities. MUDFLOW [50] discovered that the sensitive information flow pattern was different between benign and malware apps which can be utilized to do malware detection. Such data flow information was then used as features in a standard SVM to train classifiers. Gason [51] proposed a malware detection method based on efficient embedding of function call graphs with an explicit feature map, and experimentally showed it outperformed several related approaches with few false alarms. DroidSIFT [52] was semantic-based approach which extracts weighted contextual API dependency graph as program semantics to construct feature vectors. Then the feature vectors were used to train a classifier to do anomaly detection or signature based malware detection, and it has shown to be resilient for bytecode level transformation attacks.

1.3 Objectives

Similarity based malware analysis is a huge topic, and significant number of researchers have worked in this field over the past two decades as shown in above section. This dissertation tries to solve several key remaining challenges for efficient desktop malware analysis and Android malware
analysis, and explores the role of ML based malware detection in practice and meaningful ways to properly evaluate ML based malware detection systems.

- **Accurate and efficient binary similarity analysis solution**: Binary similarity analysis is an essential technique for a variety of security analysis tasks, including malware detection and malware clustering. Even though various solutions have been developed, existing binary similarity analysis methods still suffer from limited efficiency, accuracy, and usability. Entering the era of big data, effective similarity analysis solutions will be more frequently used for malware analysis. Therefore, it is necessary to create a generic and effective binary similarity analysis solution. Our first objective is to generate hashing outputs specifically for binary files that are as compact as cryptographic hashes such as SHA-2, and at the same time maintaining the accuracy with regard to overall binary similarity.

- **Effective Android malware clustering system**: Clustering has been well studied for desktop malware analysis as an effective triage method. Conventional similarity-based clustering techniques, however, cannot be immediately applied to Android malware analysis due to: (1) the excessive use of third-party libraries in Android application development; (2) the widespread use of repackaging in malware development; (3) the ambiguity between legitimate Java libraries and bogus libraries which contain malicious payloads. To address the unique challenges presented Android domain, we intend to build an effective clustering system that is specifically designed for processing Android malware.

- **Proper way to evaluate ML based malware detection system**: Existing machine learning (ML) based Android malware research obtains the ground truth by consulting AV products, and uses the same label set for training and testing. However, there is a mismatch between how ML systems have been evaluated, and the true purpose of using an ML system in practice. The goal of applying ML is not to reproduce or verify the same potentially imperfect knowledge, but rather to produce something that is better — closer to the ultimate ground truth about the apps’ maliciousness. To this end, we study the fundamental issues faced by traditional
ML based Android malware detection systems, and examine the implications of using ML based techniques for Android malware detection in practice.

1.4 Contributions

This dissertation presents an accurate and efficient binary similarity analysis solution, an effective clustering system for Android malware apps, and the implications for conducting ML based experiments to properly evaluate their performances. We summarize the major contributions in this section.

1.4.1 Accurate and Efficient Binary Similarity Analysis Tool

In this dissertation, we propose a novel structural analysis based fuzzy hashing scheme for precise and efficient binary similarity analysis. We propose a blended $n$-gram graphical feature based CFG comparison algorithm called CFGSimNG. It extracts the $n$-gram graphical features from CFGs, and measures the similarity of CFGs by comparing the graphical features encoded in numeric vectors. Based on CFGSimNG, we design and implement a fuzzy hash tool BingSim for accurate and efficient binary similarity analysis. BingSim represents binaries as fixed-size fuzzy hashes from the graphical features extracted from the corresponding CFGs. In order to comprehensively evaluate various CFG comparison algorithms, we design a clustering analysis based framework to and show that CFGSimNG is more stable, faster, and generates more accurate results compared to five state-of-the-art CFG comparison algorithms. Our comprehensive evaluation demonstrates that BingSim effectively performs malware clustering and malware detection tasks with 2865 carefully labeled malware samples in an efficient manner.

1.4.2 Specialized Android Malware Clustering System

We also design and implement an effective Android malware clustering system through iterative mining of malicious payload and checking whether malware samples share the same version of malicious payload. By reconstructing the original malicious payloads, our approach offers an effective Android malware app clustering solution along with fundamental insights into malware grouping. In order to distinguish between the legitimate Java libraries and bogus libraries which contain malicious payloads, we design a novel method to precisely remove legitimate library code
from Android apps while preserving the malicious payloads even if they are injected under popular library names. We conduct extensive experiments to evaluate the consistency and robustness of our clustering solution. Our experimental results demonstrate that our clustering approach achieves precision of 0.90 and recall of 0.75 for Android Genome malware dataset, and average precision of 0.984 and recall of 0.959 regarding manually verified ground-truth.

1.4.3 Practical Implications of ML Based Android Malware Detection

Considering the mismatch between how ML based malware detection systems were evaluated in previous research and how ML based malware detection systems should be used in practice, We suggest to evaluate machine learning outputs by checking zero-day detection results. Compared with the classical evaluation strategy that focuses on known malware detection, we believe the new evaluation strategy is more meaningful and more closely represents how an ML-based system can be beneficial in practice. We also propose to apply mislabel identification for Android malware detection and demonstrate that mislabel identification techniques can be used to prepare cleaner training datasets, or directly used for detecting zero-day malware through identification of mislabels in a noisy benign dataset. In order to verify the zero-day malware detection results, we can either verify the detection results through an updated ground truth obtained at a later time or rely on manual analysis. Through extensive experiments, we demonstrate that: (a) it is feasible to evaluate ML based Android malware detection systems with regard to their zero-day malware detection capabilities; (b) both malware prediction and mislabel identification approaches can be used to achieve verifiable zero-day malware detection, even when trained with an old and noisy ground truth dataset.

1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows. In Chapter 2, we describe a blended n-gram graphical feature based CFG comparison algorithm, which is used to create an accurate and efficient binary similarity analysis tool called BingSim. Chapter 3 summarizes the techniques challenges for using traditional similarity based clustering systems for Android malware analysis, and presents our solution for effective Android malware clustering. In Chapter 4, we examine the
issues of ML based Android malware detection systems and evaluate the effectiveness of ML based zero-day malware detection capabilities. Finally, Section 5 concludes this dissertation.
CHAPTER 2: ACCURATE AND EFFICIENT
BINARY SIMILARITY ANALYSIS

Binary similarity analysis has played an essential role in malware analysis, e.g., identifying the variants of known malware samples, evaluating the relationship between different malware families, studying the evolution of malware families, and triaging newly collected malicious samples to prioritize the new threats. It is also the most fundamental key component for conducting malware clustering analysis, which is often considered as a viable solution for large-scale malware triage. Therefore, effective and efficient binary similarity analysis is much desired for operational security analysis environments.

It is challenging to precisely and effectively compare binary codes, and the majority of existing binary similarity analysis tools are not specifically designed to evaluate binary programs. For example, security analysts propose to identify malware variants through fuzzy hashing techniques which are designed for matching file homologies. As a representative fuzzy hashing technique, ssdeep [10] has been adopted by VirusTotal [2] as one of the file properties for all of the submitted malware samples. Although ssdeep and the majority of existing fuzzy hashing tools provide efficient ways to calculate binary file properties, their capabilities to detect malware variants are limited because they consider the binary files as simple sequences of 0 and 1’s, and the low-level sequences may be substantially changed by even minor updates in the source code.

An alternative design choice to overcome the above limitation is to take the structural and semantic information of binary codes into consideration when conducting similarity analysis. Compared with low-level sequences, semantic level comparison between binary codes, e.g., by using control flow graph (CFG) level analysis, has been shown to be more robust against compilation variances [17, 53, 54].
Despite numerous efforts towards effective CFG similarity comparison, existing CFG comparison algorithms lack efficiency when performing binary similarity analysis. Graph matching is known to be computationally expensive. Even though several approximate graph isomorphism algorithms [17,55–57] have been developed over the past several decades, it is still time-consuming to compare a large number of CFGs at the same time. For instance, comparing binary A with \( m \) functions and binary B with \( n \) functions would result in \( m \times n \) pairwise CFG comparisons. Furthermore, we notice that the majority of existing CFG similarity comparison algorithms work with raw CFG structures, and rely on inefficient CFG representations for comparison. We hypothesize that the original CFG representation is not required to measure the CFG similarity if a CFG can be effectively encoded with certain representative graph features. This could result in a universal and compact graph format and make the overall comparison more efficient.

In this section, we propose a blended \( n \)-gram graphical feature based CFG comparison algorithm, called CFGSimNG, and use it as the technical basis for conducting binary similarity analysis. The new CFG comparison algorithm is designed based on the insight that the \( n \)-gram concept is applicable to represent CFGs to assess graph similarity. The \( n \)-gram concept has been extensively applied for measuring document similarity where contiguous sequences of \( n \) items are extracted from an input stream. We apply it to CFGs in a similar manner, except working with multiple input paths. Extracting \( n \)-gram graphical features from CFG structures enables us to effectively encode arbitrary CFGs in the same format. Since all CFGs in the entire binary code are encoded into the same feature space, the binary similarity is calculated in an efficient manner after extracting all the \( n \)-gram graphical features. When comparing with the state-of-the-art CFG comparison algorithms, CFGSimNG achieves the highest accuracy at the combined F-score of 0.914 for hierarchical clustering, and takes the least amount of time to complete all pairwise comparison even without caching the graph signature generation process.

To conduct binary similarity analysis, we first disassemble the target binaries to construct the CFGs of the binaries, and then extract all of the \( n \)-gram graphical features from the CFGs which are stored as binary graph signatures in a numeric vector representation. The binary similarity is then calculated by measuring the graph signature similarity. To facilitate the comparison between
binaries and efficiently store binary graph signatures, we developed BingSim to generate and compare fixed-size fuzzy hash outputs from graph signatures. In this way, we achieved high accuracy through the graphical level binary representation, and high efficiency through the compact fuzzy hash comparison. When comparing with the state-of-the-art binary similarity tools, BingSim achieves the highest accuracy at the F-score of 0.929 for singe-linkage clustering, and demonstrates higher accuracy in malware detection tasks.

2.1 Approach Overview

We illustrate the BingSim workflow in Figure 2.1. For all target binaries, we first disassemble them and construct the corresponding CFGs. We then extract the blended $n$-gram graphical features from the input CFGs and encode them as numeric vectors called graph signatures. To make it easy to use and more efficient to compare, we subsequently compress the graph signatures into fixed-size bit-vectors called fuzzy hash outputs. Finally we compare the corresponding fuzzy hash outputs to determine the similarity of input binaries. We describe the technical details of the main steps in this Section.

![Figure 2.1: The BingSim workflow](image)

2.1.1 Basic Block Type Abstraction

In order to extract representative graphical features, we need to abstract the basic blocks within binary CFGs in a meaningful way. The main objective for the abstraction is to categorize all CFG nodes into different types, and the abstracted results are then used to simulate the item “content” as used in traditional $n$-gram application scenarios.
We explore a simple basic block type abstraction approach which captures the topology of the CFG, and show in a subsequent section that such a simple type abstraction approach produces reliable results. In particular, we define the basic block types based on their number of parents (i.e., node in-degree) and number of children (i.e., node out-degree). To capture the real world situation, we experimentally analyzed a total of 93,470 binaries that were obtained from newly installed Android and Windows operation systems, and malware sharing websites like VirusShare [58]. The overall distribution of the in-degree and out-degree values are shown in Figure 2.2.

![Figure 2.2: Distribution of in-degree value, out-degree value and combined basic block types](image)

Since the majority of in-degree and out-degree values are between 0 and 3 as shown in Figure 2.2, we mainly consider in-degree values ranging from 0 to 3, and out-degree values ranging from 0 to 3. This basic block abstraction strategy results in a total of 16 unique basic block types, the combined basic block types are shown in Table 2.1, each entry in the table shows a specific basic block type, the number contained in the type string indicates the corresponding type index. Basic blocks whose in-degree are larger than or equal to 3 are considered as the same type, and
basic blocks whose out-degree are larger than or equal to 3 are considered as the same type. This won’t result in significant feature collision cases since only 1.27% of real world basic blocks have out-degree larger than 3 and only 3.67% of real world basic blocks have in-degree larger than 3.

<table>
<thead>
<tr>
<th>In-degree</th>
<th>Out-degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>≥3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B4</td>
<td>B5</td>
<td>B6</td>
<td>B7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B8</td>
<td>B9</td>
<td>B10</td>
<td>B11</td>
<td></td>
</tr>
<tr>
<td>≥3</td>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td></td>
</tr>
</tbody>
</table>

### 2.1.2 Blended n-gram Graphical Features

Inspired by the traditional n-gram concept, we consider a node (i.e., basic block) in a CFG as a single item, and define an n-gram graphical feature to be the consecutive n basic blocks that can be extracted from an input CFG. In order to accommodate the classical graphical concepts, such as nodes and edges, and consider the node connectivity and node context information at the same time, we include all k-gram \((k \in [1,n])\) features as the complete graphical feature set for the input CFG. This k-gram model (by considering all sequences from length 1 up to \(n\)) was previously referred to as blended n-gram features [59,60].

![Figure 2.3: A simple CFG and its blended N-gram features](image)

We use the simple CFG as shown in Figure 2.3 to explain the blended n-gram graphical features in more details. Each basic block is denoted with abstracted basic block types as defined
in the previous section. For example, B9 has 2 parent nodes and 1 child node. For each node in the constructed CFG, we apply the following process to extract the graphical features. For node B2, the 1-gram feature is B2 itself; the 2-gram features are B2B6 and B2B9; the 3-gram features are B2B6B9, B2B6B8 and B2B9B5. For a given n, this procedure will extract all possible blended n-gram graphical features starting from basic block B2. This step is called visiting node B2. The process for visiting one node could reach up to n-1 level children nodes. We follow this procedure for all nodes in the CFG and obtain the resulting blended n-gram graphical feature sets for the whole graph. The complete 5-gram graphical features for the CFG are presented on the right side table of Figure 2.3. Note that cycles in the CFG will not be an issue since each node in the CFG is only visited once and the visiting order of the CFG nodes makes no difference.

Increasing n-gram size can result in a larger feature space, which could provide more differentiation capabilities. On the other hand, a larger feature space also means a longer time to extract all n-gram graphical features and the extracted graph features need more resources for storage and comparison. We conduct experiments to evaluate the impact of different n-gram sizes, and empirically choose blended 5-gram as the default n-gram size considering the accuracy and efficiency (see Section 2.3.1).

With the combined basic block types defined previously, a naïve implementation for the blended 5-gram feature set would result in a feature space of 1,118,480. However, there are certain n-gram graphical features that are invalid by definition. For instance, the k-gram (k ≥ 2) features that contain in-degree of 0 basic block types (i.e., B0, B1, B2, B3) but do not start with them are invalid; the k-gram (k ≥ 2) features that contain out-degree of 0 basic block types (i.e., B0, B4, B8, B12) but do not end with them are invalid. After removing the invalid features, the blended 5-gram feature set has a smaller 118,096 legitimate entries.

2.1.3 Graph Signature Format and Comparison

We represent the n-gram graphical features in a numeric vector and consider it as a binary graph signature. Each entry in the vector indicates a specific feature, and the entry value is the corresponding feature count for the graph. In this way, both the feature content and feature size

\[^{1}\text{16 1-gram features, 16}^2 \text{2-gram features, 16}^3 \text{3-gram features, and 16}^4 \text{4-gram features, 16}^5 \text{5-gram features.}\]
are recorded. All the binary graph signatures have the same size, and the n-gram graphical feature space is determined by the n-gram size. In our implementation, we specify each feature entry as a 32-bit unsigned integer type, which we believe is large enough for practical graph sizes since it can store a maximum feature count of $2^{32}$.

We use the following cosine similarity equation to measure the basic similarity between binary graph signatures.

$$sim_{cos}(Sig_a, Sig_b) = \frac{Sig_a \cdot Sig_b}{|Sig_a| \cdot |Sig_b|}$$  \hspace{1cm} (2.1)

The initial intuition for ignoring the graph signature length during the similarity comparison is that we want to consider the CFGs that are known to be proportional in size to be the same. For example, if a CFG is duplicated multiple times, the resulting CFGs are still semantically the same as the original CFG. In addition, we also found this comparison mechanism provides an ideal foundation for meaningful signature size compression, which is used to generate the more compact fuzzy hash output as shown in the next subsection.

In practice, the feature counts for different binaries vary significantly, and completely relying on cosine similarity measure may generate misleading results. For example, comparing vectors of [1,2,3,4] and [2,4,6,8] would yield similarity score of 1.0, but the input vectors are completely different. This is due to cosine similarity only measuring the angle between the input vectors. We use $NF$ to indicate the total number of graphical features contained in the CFG, and define the following size rectification factor for two CFGs:

$$Rect_{length}(Graph_a, Graph_b) = \frac{\min(NF_a, NF_b)}{\max(NF_a, NF_b)}$$  \hspace{1cm} (2.2)

In the end, we compute the final graph signature similarity by multiplying the rectification factor $Rect_{length}(Graph_a, Graph_b)$ by the cosine similarity $Sim_{cosine}(Sig_a, Sig_b)$. According to the algorithm definition, the final similarity of two binary signatures would still be 1.0 if their graph signatures are exactly the same, and the size rectification factor is only used to penalize the input graph vectors that are sufficiently different.
2.1.4 Fuzzy Hash Generation and Comparison

The above binary graph signature representation provides a viable similarity comparison mechanism, but the signature size will increase exponentially when using larger n-gram sizes. For example, the default binary graph signature is about 461MB for blended 5-gram graphical features, which cannot reasonably be printed out and is challenging to store and compare for large-scale analysis. To make the technique easy to use and facilitate the graph signature storage and comparison process, we further compress the raw binary graph signature into a K-bit vector representation using fuzzy hashing principles.

Specifically, we pre-define a unique seed number and prepare K independent vectors with elements that are selected at random from a Gaussian distribution, and configure each random vector to be the same dimension as the raw binary graph signatures. We use $RV_i$ to represent the $i$th random vector and $Sig_b$ to represent the binary graph signature, and define the following $\text{sgn}$ function for comparing each random vector against the binary graph signature:

$$\text{sgn}(RV_i, Sig_b) = \begin{cases} 
1 & \text{if } RV_i \cdot Sig_b \geq 0 \\
0 & \text{if } RV_i \cdot Sig_b < 0 
\end{cases}$$ (2.3)

In this way, each random vector is used to create one projection for the original binary graph signature based on the dot product between the random vector and the binary graph signature, and the output of K projections is a K-bit vector. The overall random projection procedure is formally known as hyperplane locality sensitive hashing (LSH) [61, 62]. Since all the graph signatures are projected into \{0, 1\} space through the same hashing process, graph signatures with close “locality” will be projected to similar K-bit vectors. And we call the newly generated K-bit hash value a binary fuzzy hash $^2$. Given two fuzzy hash outputs $FH_a$ and $FH_b$, we apply Hamming similarity $H(FH_a, FH_b)$ to measure the similarity between the projected hash outputs, and use $\oplus$ to represent bit-wise xor operation and $|S|$ to represent the number of 1-bit in vector S. The Hamming similarity measurement $H(FH_a, FH_b)$ is defined as follows.

$^2$Note that the term “fuzzy” used here is mainly to describe the bit vector generation process, and it is unrelated to the fuzzy sets that was proposed by Lofti A Zadeh.
LSH is commonly used as a powerful technique for conducting efficient approximate nearest neighbor search with high-dimension objects. According to previous research presented in the LSH domain [61,63]: (1) cosine similarity (as used for graph signature comparison) is one type of similarity measurement that admits LSH families; (2) for any similarity function $\text{Sim}(x,y)$ that admits LSH projection, we can always obtain an LSH family $F$ that maps original objects to $\{0, 1\}$ space and has the property that the projected object similarity (e.g., Hamming similarity) is proven to correspond to the original similarity function at $\frac{1+\text{Sim}(x,y)}{2}$. Therefore, we use $\text{Sim}_\text{hash}(FH_a, FH_b)$ to estimate the original cosine similarity measurement between graph signatures through the following equation:

$$\text{Sim}_\text{hash}(FH_a, FH_b) = 2 \times H(FH_a, FH_b) - 1 \quad (2.5)$$

Increasing the number of random projection vectors can make the similarity estimation more accurate. We evaluate the impact of different $K$ values on the estimation accuracy in Section 2.3.2. The default $K$ is experimentally chosen as 256, which results in a 256-bit vector. The total number of graphical features $NF$ for each binary is represented as a 32-bit integer, thus the final fuzzy hash output is 288 bits by default. Given the $NF$ value, we can calculate the graph size rectification factor $\text{Rect}_\text{length}(\text{Graph}_a, \text{Graph}_b)$ as defined in Equation 2.2. In the end, we compute the final fuzzy hash similarity by multiplying the rectification factor $\text{Rect}_\text{length}(\text{Graph}_a, \text{Graph}_b)$ to the estimated hash similarity $\text{Sim}_\text{hash}(FH_a, FH_b)$.

2.2 Evaluation of CFG Similarity Analysis Algorithms

To evaluate the effectiveness and accuracy of the $n$-gram based CFG comparison algorithm CFGSimNG, we compare it against several representative CFG comparison algorithms. Our evaluation only focuses on the algorithms’ capability to differentiate CFG structures without considering basic block content. The auxiliary information provided by basic block content could further im-
prove CFG comparison. The abstraction process discussed in Section 2.1.1 can be extended to incorporate basic block content information and we leave this for future work.

2.2.1 Existing CFG Similarity Comparison Algorithms

A control flow graph (CFG) is an important structural representation of a binary program, and a CFG similarity analysis algorithm is the core technical component of many existing security analyses. Generic graph isomorphism analysis is a hard problem [64–67], and various techniques have been proposed for approximate CFG similarity analysis by leveraging unique properties presented in a CFG (e.g., bounded degrees).

1. **Min-cost bipartite graph matching:** Hu [17] developed an edit distance based graph isomorphism algorithm by building a cost matrix that represents the costs of mapping the different nodes in two graphs, and using the Hungarian algorithm [68] to find an optimal mapping between the nodes such that the total cost (i.e., edit distance) is minimized. Vujošević [57] iteratively built a similarity matrix between the nodes of two CFGs based on the similarity of their neighbors, and adopted the Hungarian algorithm to find the matching between the nodes in two graphs such that the resulting similarity score is the highest.

2. **Maximum common subgraph matching:** McGregor [55] designed a backtrack search algorithm to find the maximal common subgraph of two graphs. This idea has been used to design efficient CFG similarity comparison algorithms, and adopted for binary semantic difference analysis [69] and binary code search [54] scenarios. Given the maximal common subgraph output, a graph similarity score was calculated as the maximal number of common subgraph nodes divided by the number of available nodes between two graphs.

3. **k-subgraph matching:** Kruegel *et al.* [56] designed an algorithm based on k-subgraph mining. They generated a spanning tree for each node in the graph such that the out-degree of every node was less than or equal to 2, then recursively generated k-subgraphs from the spanning trees by considering all possible allocations of k − 1 nodes under the root node. Each k-subgraph was then canonicalized and converted into a fingerprint by concatenating the rows of its adjacency matrix.
4. Simulation-based graph similarity: Sokolsky [70] modeled the control flow graphs using Labeled Transition Systems. Given two CFGs, they recursively matched the most similar outgoing nodes starting from the entry nodes, and summed up the similarity of the matched nodes and edges. The overall similarity of two CFGs was then defined by a recursive formula.

5. Graph embedding: Recently, researchers have proposed to use graph embedding for CFG similarity analysis. For example, Genius [71] proposed to learn high-level feature representations from an attributed CFG (ACFG) and encoded the graphs into numerical vectors using a codebook-based graph matching approach. They used 6 block-level attributes (e.g., number of instructions) and 2 inter-block level attributes (e.g., number of offspring). Using the same features as Genius, Gemini [72] proposed a neural network-based approach to compute the graph embedding for an ACFG, and achieved better accuracy and efficiency. CFG similarity is then measured by comparing the embedded graph representation.

Our blended n-gram graphical feature based CFG comparison algorithm CFGSimNG belongs to the graph embedding category. However, CFGSimNG is different from Genius and Gemini in that it is basic block content-agnostic, and the graph embedding is always deterministic and requires no separate training process.

2.2.2 CFG Comparison Algorithm Evaluation Strategy

It is almost impossible to prepare a ground truth dataset using real world CFGs that present known similarity levels. To the best of our knowledge, the only prior work that uses any dataset to evaluate CFG similarity algorithms is that of Chan, et al. [73]. They created a ground truth CFG dataset by applying different levels of edit operations to a seed CFG, and checked whether the algorithms can output similar level of similarity differences between the generated testing CFGs and the seed CFG.

However, we observe that Chan et al.’s methodology is problematic from the following perspectives: (1) the ground truth dataset and evaluating strategy are inherently biased towards edit-distance based CFG comparison algorithms; (2) different edit operations (e.g., adding node, adding edge, deleting node, deleting edge) may have different costs: editing of a node will not
impact any existing edges, but editing of an edge will always affect two nodes, thus the testing
CFGs generated from the same levels of edit operations may present different similarity levels.

Therefore, we propose a new CFG comparison algorithm evaluation strategy by embedding
the CFG comparison algorithm in a hierarchical agglomerative clustering system as the customized
distance function\(^3\), then use the customized distance function to conduct clustering analysis for
the same ground-truth dataset, and use the overall clustering result as a performance indicator
of the corresponding CFG similarity function. Hierarchical agglomerative clustering is a bottom
up version of hierarchical clustering, in which all input items are initially considered as singleton
clusters, then for a specified distance threshold \( t \), the algorithm iteratively finds the clusters with
minimum distance and merges the clusters as long as the corresponding cluster distance \( d \) is less
than \( t \). The distance between two clusters is often referred to as “linkage” and the following three
linkage criteria are commonly used: single linkage considers the cluster distance as the minimum
distance between all the entries of two clusters; average linkage considers the cluster distance as
the average distance between all the entries of two clusters; complete linkage considers the cluster
distance as the maximum distance between all the entries of two clusters.

The rationale for evaluating different CFG comparison algorithms through hierarchical clus-
tering analysis are that: (1) the fundamental component of hierarchical clustering system is the
similarity measurement between all input items, which can be pre-calculated as a distance matrix
using each CFG comparison function. (2) when analyzing the same ground truth dataset, the only
parameter that will impact the final clustering result is the distance matrix which is controlled by
each CFG comparison algorithm. (3) the clustering analysis procedure evaluates the capability of
identifying similar items and recognizing different items at the same time.

To measure the clustering results, we adopt the measurement of precision and recall. Consid-
ering a collection of \( n \) CFGs that are reliably separated into \( m \) groups, we can represent this ground
truth reference set as \( T = T_1, T_2, ..., T_m \). For a specific distance threshold, a hierarchical agglom-
erative clustering algorithm separates the CFGs into \( c \) clusters, then the results can be represented
as \( C = C_1, C_2, ..., C_c \). The precision \( P \) and recall \( R \) are defined as:

\(^3\)distance of CFGs is calculated by the provided algorithm, distance of CFG clusters is calculated based on
individual CFG pairs according to the linkage strategy
\[ P = \frac{1}{n} \sum_{i=1}^{c} \max(|C_i \cap T_1|, |C_i \cap T_2|, ..., |C_i \cap T_m|) \]
\[ R = \frac{1}{n} \sum_{j=1}^{m} \max(|T_j \cap C_1|, |T_j \cap C_2|, ..., |T_j \cap C_c|) \]

Precision and recall measure two competing performance criteria of a clustering algorithm: the ability to separate items from different clusters, and the ability to group together items belonging to the same cluster. The extreme partition where every element forms a singleton cluster always yields a precision of 1 but will have a low recall. On the other hand, if a clustering algorithm puts all the items in a single cluster, the recall will be 1 but precision will be low. We consider the intersection point (or nearest point) between precision and recall to be the optimal clustering output. For simplicity, all the clustering results are subsequently measured with a single F-score, which is the harmonic mean of the optimal precision and recall and is calculated using the following equation.

\[ F = 2 \cdot \frac{\text{precision-recall}}{\text{precision} + \text{recall}} \]

To facilitate evaluating arbitrary CFG comparison algorithms, we plan to release the evaluation framework and the corresponding dataset. A new CFG comparison algorithm can be easily evaluated in this framework by providing a plugin that takes two CFGs as input and outputs a similarity score.

### 2.2.3 Experiment Data Preparation

To create a ground-truth CFG dataset, we compiled the latest version of Android Open Source Project code and obtained 588 ELF ARM64 binaries. We analyzed the compiled binaries, and collected all the function level CFGs that have 20 nodes, then randomly selected 5 seed CFGs from all available 20-node CFGs. We selected CFGs with 20 nodes since it typically provides enough varieties between different CFGs, and the individual CFG comparison won’t take too much time to finish for all the evaluated CFG comparison algorithms.

For each seed CFG, we apply one level of edit operation (e.g., adding node, deleting node \(^4\), adding edge, deleting edge) to create a group of artificial CFGs. Since the CFGs from each group are derived from the same seed CFG, CFGs within each group would be more similar to each other.

\(^4\)a graph node can be deleted only if it is isolated
than CFGs from different groups, either considering the “minimum edit distance” or “maximum common subgraph” perspectives. In the end, we obtained a collection of 1934 artificial CFGs in 5 different groups.

2.2.4 Evaluation Results

For algorithms proposed by Hu [17], Kruegel [56], Sokolsky [70], and Vujošević [57], we use the implementations provided by Chan [73]. We implemented McGreger’s algorithm [55] and our CFGSimNG. All the algorithm implementations evaluated in this section only consider the topology of the CFG and ignore the content of basic blocks, i.e., the content similarity between all basic block pairs are considered to be 1. Therefore, these implementations may not faithfully represent the full capability of the original designs, and the evaluation results presented in this section only reflect the algorithms’ capability of measuring similarity of CFGs’ structures, without considering the basic block content. The existing graph embedding based CFG comparison algorithms [71,72] described in Section 2.2.1 are not evaluated since they rely upon a separate training process and require specific features that are derived from concrete basic block content. If the default algorithm output is a distance score \( d \), we compute the similarity score as \( 1 - d \). The final outputs for all algorithms are normalized ranging from 0 to 1.

Table 2.2: Optimal clustering results for different algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Single linkage</th>
<th>Average linkage</th>
<th>Complete linkage</th>
<th>Combined F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hu</td>
<td>0.847</td>
<td>0.872</td>
<td>0.879</td>
<td>0.866</td>
</tr>
<tr>
<td>Vujošević</td>
<td>0.749</td>
<td>0.869</td>
<td>0.876</td>
<td>0.831</td>
</tr>
<tr>
<td>Sokolsky</td>
<td>0.367</td>
<td>0.456</td>
<td>0.501</td>
<td>0.441</td>
</tr>
<tr>
<td>Kruegel</td>
<td>0.530</td>
<td>0.530</td>
<td>0.530</td>
<td>0.530</td>
</tr>
<tr>
<td>McGreger</td>
<td>0.597</td>
<td>0.588</td>
<td>0.324</td>
<td>0.503</td>
</tr>
<tr>
<td>CFGSimNG</td>
<td>0.816</td>
<td>0.926</td>
<td>1.000</td>
<td>0.914</td>
</tr>
<tr>
<td>CFGSimNG-H</td>
<td>0.817</td>
<td>0.926</td>
<td>0.864</td>
<td>0.869</td>
</tr>
</tbody>
</table>

To avoid the bias towards a particular linkage strategy, we report the clustering results with three linkage approaches for all CFG comparison algorithms. We summarize the optimal clustering results for different algorithms in Table 2.2, and present the detailed single-linkage CFG clustering results in Figure 2.4, the average-linkage CFG clustering results in Figure 2.5, the complete-linkage
CFG clustering results in Figure 2.6. The n-gram graphical feature based CFG comparison approach is labeled as CFGSimNG, and the fuzzy hash based CFG comparison approach is labeled as CFGSimNG-H\(^5\), other approaches are labeled with corresponding main authors. The combined F-score in Table 2.2 is the average of F-scores obtained through different linkage strategies.

Figure 2.4: Single-linkage clustering results for different CFG comparison algorithms

In order to further dissect the clustering results, we separate all CFG pairs into two categories: same group CFG pairs and different group CFG pairs. Ideally, the distance outputs for the same group CFG pairs are expected to be small, and the distance outputs for different group CFG pairs are expected to be large. We present the minimum and maximum distance outputs for each group in Table 2.3 and plot the cumulative distance distribution for each algorithm in Figure 2.7.

Combining the distance range information with CFG clustering results, we can see that: (1) the algorithms proposed by Hu [17], Vujošević [57], Sokolsky [70] have very narrow overall distance ranges for All CFG pairs, thus all CFGs are quickly merged into one group during the clustering process, which results in overall high recall and low precision for the majority of provided distance thresholds. But the edit distance based CFG comparison algorithms [17, 57] generate relatively lower distance outputs for the same group CFG pairs and higher distance outputs for different

\(^5\)the detailed CFG clustering results for CFGSimNG-H is not shown since it has a very similar trend to CFGSimNG
Figure 2.5: Average-linkage clustering results for different CFG comparison algorithms

Table 2.3: Distance ranges for different CFG pairs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Same-group</th>
<th></th>
<th>Diff-group</th>
<th></th>
<th>All pairs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Hu</td>
<td>0.000</td>
<td>0.049</td>
<td>0.000</td>
<td>0.182</td>
<td>0.000</td>
<td>0.182</td>
</tr>
<tr>
<td>Vujošević</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
<td>0.213</td>
<td>0.000</td>
<td>0.213</td>
</tr>
<tr>
<td>Sokolsky</td>
<td>0.000</td>
<td>0.258</td>
<td>0.000</td>
<td>0.258</td>
<td>0.000</td>
<td>0.258</td>
</tr>
<tr>
<td>Kruegel</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
<td>0.000</td>
<td>1.000</td>
</tr>
<tr>
<td>McGreger</td>
<td>0.000</td>
<td>0.905</td>
<td>0.000</td>
<td>0.800</td>
<td>0.000</td>
<td>0.905</td>
</tr>
<tr>
<td>CFGSimNG</td>
<td>0.007</td>
<td>0.724</td>
<td>0.181</td>
<td>0.921</td>
<td>0.007</td>
<td>0.921</td>
</tr>
<tr>
<td>CFGSimNG-H</td>
<td>0.015</td>
<td>0.978</td>
<td>0.328</td>
<td>1.000</td>
<td>0.015</td>
<td>1.000</td>
</tr>
</tbody>
</table>

- group CFG pairs, therefore they still generate overall good F-score outputs. (2) the algorithms proposed by Kruegel [56] and McGreger [55] provide broader distance ranges, but for same group CFG pairs and different group CFG pairs at the same time. This makes the precision and recall from different clustering thresholds slowly intersect with each other or never intersect at all, which results in an overall poor F-score. (3) CFGSimNG and CFGSimNG-H both provide a very good distance range. Figure 2.7 also demonstrates that they can clearly separate the majority of same group CFG pairs and different group CFG pairs, for example, when choosing distance threshold around 0.55 for CFGSimNG and choosing distance threshold around 0.70 for CFGSimNG-H. Therefore, they both generate very good F-score outputs.
Figure 2.6: Complete-linkage clustering results for different CFG comparison algorithms

We believe the high performances for CFGSimNG and CFGSimNG-H are mainly because: the differences between n-gram graphical features are always proportional to CFGs' structural differences. For example, the same group CFGs will have more similar n-gram graphical features and the different group CFGs will have more different graphical features, which subsequently leads to the desired differentiation capabilities.

In summary, the distance calculation results for CFGSimNG show the best separation capabilities between similar CFG pairs and different CFG pairs, while existing CFG comparison algorithms either have very small distance output ranges or have almost the same distance ranges between same group CFG pairs and different group CFG pairs. This demonstrates that CFGSimNG is more stable and has a more balanced capability for recognizing similar CFGs and identifying different CFGs at the same time.

2.2.5 Overall Performance

For each algorithm, we record the time taken (in seconds) to finish the similarity calculation for all CFG pairs and present the corresponding results in Table 2.4. Since the hierarchical clustering algorithm has \( n^2 \) complexity, the prepared 1934 CFGs would result in 1869211 pairwise CFG comparisons. Note that all the algorithm implementations take two CFGs as input and output...
a similarity score, which means the graph signatures for CFGSimNG and fuzzy hash outputs for CFGSimNG-H are generated 1869211 times during the evaluation. This makes the CFGSimNG-H approach dramatically slower than other approaches. However, if the graph embedding process is cached, CFGSimNG only needs 60.8s to finish the graph signature generation and all pairwise comparisons, and CFGSimNG-H only needs 27.7s to finish the fuzzy hash generation and all pairwise comparisons, which is more efficient than all existing approaches.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hu</td>
<td>755.6s</td>
</tr>
<tr>
<td>Vujosevic</td>
<td>1788.0s</td>
</tr>
<tr>
<td>Sokolsky</td>
<td>483.1s</td>
</tr>
<tr>
<td>Kruegel</td>
<td>321.8s</td>
</tr>
<tr>
<td>McGregor</td>
<td>2542.1s</td>
</tr>
<tr>
<td>CFGSimNG</td>
<td>461.4s</td>
</tr>
<tr>
<td>CFGSimNG (cache)</td>
<td>60.8s</td>
</tr>
<tr>
<td>CFGSimNG-H</td>
<td>10.5h</td>
</tr>
<tr>
<td>CFGSimNG-H (cache)</td>
<td>27.7s</td>
</tr>
</tbody>
</table>

2.3 Evaluation of Binary Similarity Analysis Tools

We first analyze the effectiveness of using BINGSim to conduct binary similarity analysis, and compare it against the following existing binary similarity comparison solutions: ssdeep v2.14.1 [10] and BinDiff v4.3.0 [74]. Other previously proposed binary similarity analysis tools were not evaluated because they were either not maintained [75] (i.e., not working with majority of binaries), or never publicly released [69]. In order to exclude the impact of CFG construction accuracy brought
by different disassemblers, BingSim was implemented as an IDA Pro plugin. The current prototype implementation of BingSim uses IDA Pro v6.8 to process the target binaries and construct corresponding CFGs. The same version of IDA Pro is also used by BinDiff.

### 2.3.1 Binary Graph Signature Accuracy

For simplicity, we call the original graph signature based binary similarity analysis tool BingSim-O and the fuzzy hash based binary similarity analysis tool BingSim. Since BingSim was built on top of BingSim-O, we first check the overall accuracy of BingSim-O and evaluate the accuracy impact of different $n$-gram sizes.

Like the evaluation strategy presented in Section 2.2.2, we decided to prepare a ground truth binary dataset, then embed the BingSim-O as the customized distance function for a hierarchical clustering system, and use the clustering results as the accuracy indicator for BingSim-O. It is ideal to evaluate the accuracy impact of different $n$-gram sizes through clustering analysis since the $n$-gram size is the only configurable parameter for BingSim-O. It can be used to control graph signature sizes, which subsequently impacts the CFG embedding accuracy and final clustering accuracy.

We prepared the ground truth dataset by collecting a set of desktop malware samples with known family names, and assume that the binaries with the same family name are more similar to each other than binaries from different families. Because the labeled malware datasets used by previous research are either discontinued or only released as a hash list, we decided to prepare our own labeled malware ground truth dataset. In the end, we collected 2865 recent desktop malware samples from VirusShare [58], every sample was consistently labeled by at least 25 AV products listed on VirusTotal [2]. The resulting malware ground truth dataset contains 8 different malware families and is shown in Table 2.5.

<table>
<thead>
<tr>
<th>Malware Family</th>
<th>Size</th>
<th>Malware Family</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstallRex</td>
<td>1115</td>
<td>OutBrowse</td>
<td>615</td>
</tr>
<tr>
<td>MultiPlug</td>
<td>384</td>
<td>DomaIQ</td>
<td>184</td>
</tr>
<tr>
<td>LoadMoney</td>
<td>173</td>
<td>Linkular</td>
<td>164</td>
</tr>
<tr>
<td>InstallCore</td>
<td>127</td>
<td>DownloadAdmin</td>
<td>103</td>
</tr>
</tbody>
</table>
Table 2.6: BINGSim-O Accuracy with different n-gram sizes

<table>
<thead>
<tr>
<th>N-gram Size</th>
<th>Single Linkage</th>
<th>Average Linkage</th>
<th>Complete Linkage</th>
<th>Combined F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.899</td>
<td>0.907</td>
<td>0.882</td>
<td>0.896</td>
</tr>
<tr>
<td>3</td>
<td>0.905</td>
<td>0.908</td>
<td>0.881</td>
<td>0.898</td>
</tr>
<tr>
<td>4</td>
<td>0.912</td>
<td>0.908</td>
<td>0.882</td>
<td>0.901</td>
</tr>
<tr>
<td>5</td>
<td>0.934</td>
<td>0.911</td>
<td>0.909</td>
<td>0.918</td>
</tr>
<tr>
<td>6</td>
<td>0.935</td>
<td>0.917</td>
<td>0.917</td>
<td>0.923</td>
</tr>
</tbody>
</table>

We also use precision and recall to evaluate the clustering outputs, the overall clustering results using different n-gram sizes are shown in Table 2.6. We can see from the table that 2-gram already generates decent results, and further increasing n-gram size only slightly increases the optimal F-score result. Considering both of the accuracy and efficiency, we choose 5 as the default n-gram size for generating the original binary graph signatures.

2.3.2 Graphical Fuzzy Hash Accuracy

Since the binary fuzzy hash representation is significantly smaller than the raw graph binary signature, it is critical for the fuzzy hash to generate comparable similarity measurement results. We evaluate the fuzzy hash similarity accuracy by comparing the binary fuzzy hash similarity results against the raw binary graph signature similarity results. For this experiment, we randomly selected 1000 native binaries from the Windows platform.

We first extract graph signatures for the binaries and calculate the graph signature similarities (i.e., the BINGSim-O similarity results); then generate the binary fuzzy hashes and calculate the corresponding fuzzy hash similarities (i.e., the BINGSim similarity results). For the same binary pair, we use δ to represent the similarity difference between graph signature similarity and fuzzy hash similarity. We collect the list of all similarity differences using the following different K’s: 64, 128, 256, 512, 1024. The mean and standard deviation of the similarity differences for all K sizes are plotted in Figure 2.8.

From the resulting graph, we can see that: (1) the mean of the similarity differences for all cases are less than 0.06; (2) increasing the size of K reduces the average and standard deviation of the similarity differences, but it does not improve much after K reaching 256. Therefore, we choose the default K value to be 256 for generating fuzzy hash outputs. The histogram of the similarity
differences when selecting $K$ to be 256 is shown in Figure 2.9. Even though the average similarity differences between BingSim-O and BingSim are still above 0, Figure 2.9 shows that the majority of similarity differences are very close to 0.

2.3.3 Similarity Tool Output Consistency

We evaluated the output consistency for all binary similarity analysis tools (BingSim, BinDiff, ssdeep). For this experiment, we randomly selected 3000 native binaries from the Windows platform.

We first check whether the binary similarity analysis tools can detect binaries that are known to be the same. We create a duplicate for each selected binary, then compared each of the original binary against its duplicated version. BingSim and ssdeep detected all 3000 binary pairs with 1.0 similarity. However, we noticed that the pairwise similarity outputs for BinDiff are ranging from 0.184 to 0.993 with average similarity of 0.929, and never reached a similarity of 1. The majority of the similarity outputs for BinDiff are around 0.631 and 0.950. After checking the BinDiff manual, we note that the final similarity outputs generated by BinDiff were multiplied by a “confidence” score, which indicates whether the CFG match is produced by a weak or strong matching algorithm.
Thus, even though two binary files are identical and all functions are perfectly matched, BinDiff may still generate a low final similarity score due to the weak matching algorithms used.

We also checked whether the binary similarity analysis tools can generate consistent similarity output if we switch the input order. We made a pairwise comparison for all the selected binaries. Given a pair of binaries, we first calculated their similarity scores by sorting them in alphabetically increasing order, then calculated their similarity scores using reverse order. BingSim and ssdeep produced the same similarity outputs for all the binary pairs. However, for BinDiff, only 47.9% of the 4998500 binary pairs generate the same similarity outputs, the rest of the binary pairs have similarity differences ranging from 0.001 to 0.192. We suspect the output differences are because the final matching between functions and basic blocks are not globally optimal, thus generating inconsistent outputs if provided different input orders. A simple fix without significantly changing the existing algorithm is to sort the inputs internally before conducting the similarity analysis. A more advanced fix is to design an algorithm that always finds the globally optimal match between all functions and all basic blocks, which may increase the matching algorithm complexity though. In order to obtain deterministic results for subsequent evaluation, we always use the alphabetically increasing input order for BinDiff.

2.3.4 Similarity Tool Accuracy Analysis

We further embed all binary similarity analysis tools (BingSim, BinDiff, ssdeep) into the hierarchical clustering system, and use the clustering outputs to evaluate the similarity measurement accuracy of all the tools. For this experiment, we use the 2865 labeled malware samples as prepared in Section 2.3.1.

For the three similarity analysis tools, we summarize the optimal clustering results with regard to different clustering strategies in Table 2.7, and plot the overall clustering results with different binary similarity analysis tools in Figure 2.10. We can see from the table that, BingSim generates the highest F-score of 0.929 for single linkage clustering analysis, which is very close the 5-gram based BingSim-O clustering outputs shown in Table 2.6. BinDiff produces similar results with F-score of 0.883, but ssdeep only outputs an F-score of 0.690. Because ssdeep operates at the binary stream level, it won’t be able to identify a significant number of semantically similar
binaries; BinDiff and BingSim both operate at the CFG level, thus identify a larger number of similar binaries. The F-score differences between BinDiff and BingSim are mainly due to the different similarity calculation logic of the tools.

Table 2.7: Optimal clustering results for different tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Single Linkage</th>
<th>Average Linkage</th>
<th>Complete Linkage</th>
<th>Time Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssdeep</td>
<td>0.690</td>
<td>0.690</td>
<td>0.689</td>
<td>2.7m</td>
</tr>
<tr>
<td>BinDiff</td>
<td>0.883</td>
<td>0.883</td>
<td>0.883</td>
<td>166.4m</td>
</tr>
<tr>
<td>BingSim</td>
<td>0.929</td>
<td>0.903</td>
<td>0.909</td>
<td>0.9m</td>
</tr>
</tbody>
</table>

We also evaluated the time taken to conduct the experiment with each tool. Since BinDiff does not have an intermediate “signature” representation for binary CFGs, we only calculated the time taken to finish all the pairwise computations. And for pairwise comparison with BinDiff, we converted all the input binaries to “BinExport” format. The overall pairwise comparison time for each tool is shown in the last column in Table 2.7. We can see that BingSim achieves higher efficiency than ssdeep, and BinDiff is dramatically slower than other tools. The main reason for the high efficiency of BingSim is that the fuzzy hash is essentially a bit-vector, which is more CPU friendly than the ssdeep hash comparison.

It is worth mentioning that various previous malware clustering systems already demonstrate very promising results. For example, using different datasets, Malheur [41] reported F-score of 0.95, BitShred [8] showed F-score of 0.932, and FIRMA [76] claimed F-score of 0.988 in their papers. As a reference, we choose to conduct experiments with BitShred, which is a state-of-the-art malware clustering tool using static features. Since BitShred only adopts the single-linkage clustering strategy, we plot the single-linkage clustering results for all tools in Figure 2.10. From the graph, we can see that BitShred reaches the optimal F-score of 0.885. Overall, BingSim generates best clustering results (e.g., F-score of 0.929) than other tools if using single-linkage clustering strategy.

Figure 2.10 also shows that the recall results for BingSim and BitShred at a distance threshold of 0 are above 0.650, while the recall results for ssdeep and BinDiff at this threshold are 0. This is because BingSim and BitShred correctly identify a significant number of binary pairs with a
similarity of 1.000, while both ssdeep and BinDiff can not identify any of such binary pairs. We further notice that the majority precision values (e.g., with distance thresholds of [0.000, 0.995]) for ssdeep are 1.000, which means all the binary pairs that are identified as similar (i.e., similarity score larger than 0.005) are indeed similar. However, at the same time, the corresponding recall values for ssdeep are less than 0.365, which indicates that ssdeep couldn’t recognize a significant number of binaries that are known to be similar regardless of distance thresholds. This is in line with our practical usage experience with ssdeep.

### 2.3.5 Malware Detection Analysis

Binary similarity analysis can be easily used for malware detection purpose. For instance, it is a common sense for security analysts that, if certain malware samples have been thoroughly analyzed, then we can use the known malicious samples to detect the variants from the same family. One of the most popular existing tools for this purpose is ssdeep, which was used by VirusTotal [2]
since 2012. We try to formally evaluate the accuracy for such a malware detection approach in this section.

We used the 2865 labeled malware dataset as a malicious ground truth dataset, and used the collected binary samples from fresh Android and Windows operation systems as benign ground truth dataset. Next, we randomly selected $X$ samples from each malware family for training, and use the rest of the malicious samples (i.e., $2865 - 8 \times X$ binaries) and all the benign dataset for testing. In the end, we collected 60557 benign binaries, the benign dataset composition is presented in Table 2.8. Notice that similarity based malware detection aims to detect variants from known malware families, all the unlabeled potentially malicious samples are not used.

<table>
<thead>
<tr>
<th>Binary Category</th>
<th>Size</th>
<th>Binary Category</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 32-bit</td>
<td>14480</td>
<td>Windows 64-bit</td>
<td>13152</td>
</tr>
<tr>
<td>Android 32-bit</td>
<td>15854</td>
<td>Android 64-bit</td>
<td>17071</td>
</tr>
</tbody>
</table>

We employ precision and recall to evaluate the malware detection results. Precision is defined as the ratio of the detected samples that are indeed malicious, and recall is defined as the ratio of malicious samples that are detected. In particular, we utilize the optimal similarity thresholds obtained from single-linkage clustering analysis in Section 2.3.4 as the binary “maliciousness” decision criteria for the three similarity analysis tools. Therefore, the similarity thresholds for ssdeep, BinDiff, and BINGSIM are configured as 0.005, 0.738 and 0.832, respectively.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Training binary size (X)</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssdeep</td>
<td>precision</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>recall</td>
<td>0.307</td>
<td>0.524</td>
<td>0.593</td>
<td>0.620</td>
</tr>
<tr>
<td>BinDiff</td>
<td>precision</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>recall</td>
<td>0.842</td>
<td>0.901</td>
<td>0.926</td>
<td>0.951</td>
</tr>
<tr>
<td>BINGSIM</td>
<td>precision</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>recall</td>
<td>0.883</td>
<td>0.960</td>
<td>0.971</td>
<td>0.991</td>
</tr>
</tbody>
</table>

In order to avoid random errors, we conducted the experiments 3 times for each specific $X$, and calculate the average precision and recall as the final measurement for each $X$. Table 2.9 shows the
overall malware detection results with regard to different malware training sizes. We can see that: (1) all binary similarity analysis tools returned precision of 1.00 which means no benign samples are detected as malicious; but the recall for ssdeep can only reach 0.620 when selecting 15 training samples from each family, which means ssdeep suffered significantly for false negatives; (2) even only selecting one sample from each malware family, BingSim and BinDiff can already achieve detection recall higher than 0.842 which means CFG level comparison can significantly increase the likelihood of identifying malware variants of known malware families. Overall, increasing the training binary size further increases the detection accuracy of all three approaches, but BingSim consistently maintained higher recall results for any training sizes.

2.3.6 Obfuscation State Analysis

Since malware authors frequently employ various obfuscation techniques to avoid being detected by anti-virus engines, it is critical to identify obfuscation state of the analyzed binaries and the practical impact of obfuscation techniques. For the labeled 2865 malware samples, we use PEid, ExeInfo [77], and DIE [78] to identify the packer information for each binary. We noticed that the packers identified by PEid and DIE are subsets of the ExeInfo outputs after normalizing the packer names. Table 2.10 describes the normalized packer information identified by ExeInfo. As shown in the table, the most commonly used packers for the labeled malware dataset are InstallMate [79], Nullsoft Installer [80], and Inno Setup [81]; and only Nullsoft Installer (with different versions) was used for multiple families. Even though these packers can compress the original binaries, they are mainly used to create software installation bundlers. Such installers or bundlers can pack multiple files at the same time, the execution of the packed binary is the installation process, e.g., creating appropriate directories and copying the original executable to desired file system locations. Thus, they do not belong to the traditional “packers” known in the malware analysis community that are designed to obfuscate the code or deter reverse engineering of the binary.

We further analyzed the malware binaries with a machine learning based packer classification tool developed by Perdisci et al. [82]. Using the training dataset provided by Perdisci et al., we identify 521 unpacked binaries, which consist of 381 MultiPlug samples, 93 DownloadAdmin Samples, and 47 LoadMoney samples. The rest of the labeled 2865 malware samples are all classified
Table 2.10: Packer identification results

<table>
<thead>
<tr>
<th>Packers Identified</th>
<th>Size</th>
<th>Major Malware families</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstallMate</td>
<td>1115</td>
<td>InstallRex</td>
</tr>
<tr>
<td>Nullsoft Installer</td>
<td>950</td>
<td>Outbrowse, Linkular, DomaIQ</td>
</tr>
<tr>
<td>Inno Setup</td>
<td>127</td>
<td>InstallCore</td>
</tr>
<tr>
<td>UPX</td>
<td>30</td>
<td>LoadMoney</td>
</tr>
</tbody>
</table>

As packed. To evaluate the practical impacts of packing tools on binary similarity analysis tools, we then manually packed the 521 unpacked binaries with Inno Setup, Aspack [83], and UPX [84] using their default configurations. For the 521 unpacked samples, we obtained 135460 binary pairs. Based on their original family label information, we separated the 135460 binary pairs into 77749 same family pairs (shown as “same-group” in Table 2.11) and 57711 different family pairs (shown as “diff-group” in Table 2.11). Table 2.11 presents the median\(^6\) similarity scores calculated by different tools after applying the packing tools.

Table 2.11: Similarity calculation impacts for similarity analysis tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Different packing tools</th>
<th>None</th>
<th>Inno</th>
<th>Aspack</th>
<th>UPX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssdeep</td>
<td>same-group</td>
<td>0.000</td>
<td>0.460</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>diff-group</td>
<td>0.000</td>
<td>0.430</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>BinDiff</td>
<td>same-group</td>
<td>0.666</td>
<td>0.991</td>
<td>0.993</td>
<td>0.992</td>
</tr>
<tr>
<td></td>
<td>diff-group</td>
<td>0.079</td>
<td>0.991</td>
<td>0.993</td>
<td>0.959</td>
</tr>
<tr>
<td>BINGSIM</td>
<td>same-group</td>
<td>0.801</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>diff-group</td>
<td>0.036</td>
<td>1.000</td>
<td>1.000</td>
<td>0.944</td>
</tr>
</tbody>
</table>

From Table 2.11, we can see that: (1) BinDiff and BINGSIM both present strong similarity differences between the same family binary pairs and different family binary pairs; ssdeep only identifies 5% of the same family binary pairs, and the majority of outputs are similarity of 0. (2) after packing binaries, the similarity differences between same family binary pairs and different family binary pairs become little. This is unsurprising because the CFGs of the packed binaries typically belong to the decompressing routines instead of the original functionalities. (3) as shown by ssdeep outputs, the packed binaries for Inno Setup demonstrate higher overall similarity than binaries generated by traditional packers. This is because Inno Setup compresses the original

---

\(^6\)median is used to summarize the similarity score distributions since it is more resistant to the outliers
binaries altogether while Aspack and UPX process the program code, data, and resources sections individually.

As known [85] in the security community, this experiment confirms again that the obfuscation tools make it challenging for static analysis based malware processing since the comprehensible structure of the packed binaries are mainly the decompression or decryption routines. Nevertheless, a functioning similarity analysis tool would still be useful to reliably identify the common packers because of the shared decompressing routines. To demonstrate this, we consider the binaries packed by the same tool as one family and mix the 1563 packed binaries, then conduct clustering analysis using three similarity analysis tools. Regardless of the clustering mechanisms, BinDiff and BINGSIM both generate an F-score of 1.0 while ssdeep only generates an F-score of 0.667.

2.4 Limitations

2.4.1 CFG Construction Accuracy

Binary reverse engineering and CFG construction are challenging in practice, especially for identifying function boundaries and resolving indirect jumps. The overall procedure has been widely discussed in the literature [86–90], and in this work, we consider CFG construction as an independent preprocessing step for our analysis, and rely on existing methods to disassemble binary codes and construct CFGs.

BINGSIM is currently implemented as an IDA Pro plugin, and it can benefit from the development of advanced disassembly techniques. It can be further implemented as a standalone similarity analysis tool using other existing disassembly engines, such as angr [91], and be leveraged to quickly identify the variants of known binary codes.

2.4.2 Obfuscation and Evasion Techniques

Malware samples are often packed in recent years to evade signature-based malware detection tools [92]. Even worse, malware authors can apply multiple layers of packing, or employ advanced packers that dynamically decrypt original code on-the-fly or interpret instructions in a virtualized environment. In this work, our main objective is to design an efficient and accurate binary similarity
analysis tool, and we consider a comprehensive solution [93-95] for solving binary obfuscation as an orthogonal problem.

We believe BingSim is still helpful in practice for the following reasons. (1) Lots of real world binaries are still unpacked, especially for adware or PUP programs. BingSim can be used to quickly filter out similar binaries that have been processed before, or used for triaging a large amount of unprocessed binaries (even packed ones) by grouping similar instances together. For this purpose, traditional cryptographic hash and existing similarity analysis solutions are not effective.

(2) Comparing to low level binary sequences, it will be more difficult to add randomness to the CFG structure for the packed binaries. For instance, dead code can be removed during the CFG construction procedure. Thus the packed binaries would always share certain deobfuscation routines, which can be viewed as the “signature” of the packers. As shown in Section 2.3.6, an accurate binary similarity analysis tool can be used to identify the shared decompressing routines, which would be useful for handling packed binaries by forwarding them to the corresponding deobfuscation routine.

2.4.3 Feature and Signature Collision

We consider the potential attack scenario by generating similar graph features for different CFGs. According to the design in Section 2.1.1, feature collision happens when: (1) nodes indegree or outdegree is larger than 3; (2) nodes with different content but same shape. The first type of collision is rare (e.g., 3.67% of nodes) in real world binaries; the second type of collision is largely alleviated by recording the node context (i.e., n-gram features) and graphical feature counts. Thanks to the N-gram feature extraction technique, the resulting feature differences are always proportional to the input structural differences.

The proposed CFG similarity analysis algorithm CFGSimNG compares CFG graph signatures by measuring the overall cosine similarity of graph signatures and the relative CFG sizes. Since the graph signature is mainly a summary of all features contained in CFG, it is theoretically possible for binaries with different CFGs to generate similar graph signatures. Nevertheless, the random signature “collision” for overall binary CFGs seems to be rare in practice, as demonstrated by our binary similarity based malware detection experiment in Section 2.3.5.
To further reduce the collision possibilities, we can increase the graphical feature space by incorporating certain basic block content information into the type abstraction process, such as presence of string or numeric constants, number of instructions etc.

2.5 Summary

In this chapter, we proposed a blended n-gram based CFG comparison algorithm CFGSimNG, which compares CFGs using n-gram graphical features. In order to compare with existing CFG comparison algorithms, we designed a clustering analysis based framework, and evaluation results show that CFGSimNG is more effective and efficient compared to existing techniques. Based on CFGSimNG, we developed a graphical comparison based fuzzy hash tool BingSim for binary similarity analysis. We further empirically demonstrated that BingSim performs efficient and effective similarity analysis based malware detection and malware clustering.
CHAPTER 3: PAYLOAD MINING
BASED ANDROID MALWARE CLUSTERING

Triaging is an important step in malware analysis given the large number of samples received daily by security companies. Clustering, or grouping malware based on behavioral profiles is a widely-studied technique that allows analysts to focus their efforts on new types of malware. Multiple static [8, 96], dynamic [42, 97], and hybrid [98] analysis based clustering techniques have been proposed in the desktop malware domain.

With the rapid growth of Android smart devices, malicious Android apps have become a persistent problem. Security companies receive a list of (potential zero-day) malware on a daily basis [28]. Those apps that present certain suspicious behaviors but are not detected by any existing anti-virus scanners need to be further analyzed manually. Conducting clustering on those incoming malware apps can allow the analysts to triage their tasks by (a) quickly identifying malware that shares similar behaviors with known existing malware so they may not allocate much resources to it; and (b) selecting a few representative apps from each new malware cluster to prioritize their analysis.

We often observe that existing approaches to group Android malware based on their behaviors have provided limited capabilities. For example, existing Android malware detection products may report a family name for a detected sample; however, samples from one family can have multiple different versions of malicious code segments presenting significantly different behaviors. Therefore, the malware family information provided by AV products can be incomplete to describe crucial malicious code segments of Android malware.

---

This chapter was published in The 20st International Symposium on Research in Attacks, Intrusions, and Defenses. Permission is included in Appendix.
Existing overall similarity analysis based clustering systems cannot be immediately applied for Android malware clustering because the malicious code segments often constitute only a small fraction of an Android malware sample. In desktop malware clustering, the static or dynamic features are first extracted from target samples. Then a clustering algorithm (e.g., hierarchical agglomerative clustering) is applied to group the samples such that samples within the same resulting group share a high level of overall similarity. However, we note that overall similarity analysis performs poorly in Android malware clustering because of two common practices in Android malware development.

The first practice is repackaging. Malware writers may embed the malicious code inside an otherwise legitimate app, in which case the real malicious code segment is likely to be small compared to the original benign app. Our analysis shows that the ratio of the core malicious code segments to the entire app for a collection of 19,725 malware samples is between 0.1% and 58.2%. Given the small percentage of malicious code segments, the conventional clustering approach that is based on overall code similarity will not work well. For example, two malicious samples from different families can be repackaged based on the same original benign app, thus presenting a high level of overall similarity. Likewise, Android malware variants with the same malicious code of one family can be repackaged into different original benign apps, thus presenting a low level of overall similarity.

Another practice is utilizing shared library code. Android apps often include a variety of third-party libraries to implement extra functionalities in a cost-effective way. If the library code size is too large compared to the rest of the app, samples from different malware families may be clustered together simply because they share the same libraries. We measured the library code proportion of the 19,725 malware samples. For 13,233 of the samples that used at least one legitimate library, we found that the average library code ratio was 53.1% in terms of number of byte code instructions. This means a large portion of an Android app belongs to libraries. One approach to prevent those libraries from “diluting” the malicious code segments is to use a whitelist [23, 33, 99–101] to exclude all library code. However, previous work leverages only the names of libraries while building a whitelist as opposed to the content of libraries. We observed that malware authors injected their ma-
licious code under popular library names, such as `com.google.ssearch`, `com.android.appupdate`, `android.ad.appoffer`, and `com.umeng.adutils`. Consequently, naïve whitelisting approaches inadvertently remove certain malicious payloads together with the legitimate library code from analysis. We found that about 30% of our analyzed Android malware families disguise their malicious payload under popular library names.

Due to the above two reasons, directly applying overall similarity analysis on Android apps will not be effective for Android malware analysis. A major challenge is to precisely identify the malicious code segments of Android malware. For simplicity, we refer to the core malicious code segments of Android malware as the **malicious payload**. A payload can be an added/modified part of a repackaged malware app, or the entire code of “standalone” malware app excluding legitimate library code.

In this section, we propose an Android malware clustering approach through iterative mining of malicious payloads. The main idea of our Android malware clustering approach is to mine the malicious payloads from malware samples, and group malware samples by checking if they share the same version of malicious payloads. Particularly, we design a robust method to precisely exclude legitimate library code from Android malware while retaining the malicious code segments even if they are injected under popular library names. We then extract candidate payloads by identifying the shared code segments between each malware app pair, and group the extracted payloads based on their code similarity where each candidate payload group comprises one version of shared code segments. We further iteratively examine the candidate payload groups that are most likely to be malicious payloads and eventually group malware samples based on selected payload clusters and payload-to-app association information. In this way, Android malware samples are clustered together based on the shared similar version of **malicious payloads** instead of the overall similarity of the entire applications, which may include original benign code and legitimate library code.

## 3.1 Overview of Android Malware Clustering System

Rather than directly conducting overall similarity analysis between Android malware samples, we first design a solution to precisely remove legitimate library code from Android apps. We consider the shared code segments (excluding legitimate library code) between the analyzed Android apps
as the candidate payload, and find all of the input Android apps through pairwise intersection analysis. For a group of $n$ apps, each input app will contribute to $n - 1$ versions of candidate payloads.

After extracting all candidate payloads, we conduct traditional clustering analysis on all candidate payloads to group similar ones together. Based on several key insights that are learned from analyzing candidate payload clustering results, we designed an effective approach to iteratively mine the payload clusters that are most likely to be malicious, and make sure that each input app will only contribute one version of a malicious payload. Finally, we use the identified malicious payload clusters and payload-to-app association information to group the input Android malware apps. We describe this process in more details below.

Figure 3.1 illustrates the overview of the clustering analysis system with five malware samples.

1. **Library code removal**: We convert malware samples into fingerprint representation, and design an effective approach to precisely remove legitimate library code from each app fingerprint. We denote the library-removed app fingerprints as $fp_1$, $fp_2$, $fp_3$, $fp_4$, and $fp_5$ accordingly.

2. **Candidate payloads extraction**: We conduct a pairwise intersection analysis to extract all shared code segments (e.g., candidate payloads) between input apps. Relying on the app fingerprint representation, we create candidate payload fingerprints, and record the payload-
to-app association information. For example, fp1-2 indicates that this candidate payload is extracted from malware sample 1 and 2.

3. **Candidate payloads clustering:** We then perform hierarchical clustering on all candidate payloads with a predefined clustering similarity threshold $\theta$, e.g., the candidate payload fingerprints fp1-2, fp1-3, and fp2-3 are grouped together as the largest payload cluster based on the overall payload similarity.

4. **Malicious payload mining:** After removing legitimate libraries, similar malicious payloads extracted from samples in the same malware family will become more popular due to the "legitimate" reason of code reuse. Therefore, we design an iterative approach to mine the popular payload clusters from the clustering results, which are more likely malicious payloads. For instance, the candidate payload cluster containing fp1-2, fp1-3, and fp2-3 is selected as the most popular cluster. To ensure that each input app only contributes one version of final malicious payload, we simultaneously update the remaining payload clusters. e.g., fingerprints fp1-4, fp1-5, fp2-4, fp2-5, fp3-4, and fp3-5 are then skipped because malware sample 1, 2 and 3 have already been "used".

5. **Malicious samples grouping:** We group the original Android samples based on payload mining results and payload-to-app association information such that the samples within each app cluster contains the same version of the malicious payload. For example, malware samples 1, 2, and 3 are grouped together based on the selected candidate payload cluster containing fp1-2, fp1-3, and fp2-3.

### 3.2 App Fingerprint Representation and Utilization

As we can see from Section 3.1, the clustering system requires an effective fingerprint representation to denote input Android apps and candidate payloads. Ideally, the fingerprint needs to be constructed from the code segments of the input app and support two fundamental operations: precisely removing legitimate code, correctly extracting shared app code.

\[1\text{Further intuition explanation and popularity criteria are included in Section 3.3.}\]
Based on these requirements, we decided to represent Android apps as bit-vector fingerprints, by encoding the features that are extracted from app code through feature hashing [8,15,38]. The value of each bit in the generated fingerprint is either 0 or 1, indicating whether the corresponding app has a specific feature or not.

This bit-vector format enables us to precisely remove legitimate library code (Section 3.2.2), extract shared code segments (Section 3.2.3), and reconstruct the original malicious payload (Section 3.2.4) by utilizing the bit manipulation capability.

### 3.2.1 Fingerprint Generation and Fingerprint Comparison

In this work, we use the $n$-gram sequence of Dalvik bytecode to denote an Android app feature, and use a bit-vector fingerprint to represent the extracted features. The overall fingerprint generation process is shown in Figure 3.2.

![Fingerprint Generation Diagram](image)

**Figure 3.2: Overall fingerprint generation procedure**

For each Android app, we first use Dexdump [102] to disassemble `classes.dex` into Dalvik bytecode, then preprocess the Dalvik bytecode sequences to only include the major distinctive information and extract the $n$-gram features from the preprocessed bytecode sequences. We follow a similar approach to extract the distinctive information (e.g., bytecode opcode) for feature construction as Juxtapp [22]. Since the feature space is vital to support the key operations designed in this work, we decided to increase the feature space by including more generic but meaningful information from each bytecode instruction. The major distinctive information is separated into 4 categories and summarized in Table 3.1. Besides the feature differences shown in Table 3.1, we extract the $n$-gram features at the function level, while Juxtapp extracts $n$-gram features at the basic
block level. For simplicity, we only show the Dalvik bytecode opcode sequences as the distinctive instruction information in Figure 3.2.

Table 3.1: Major feature categories and main differences comparing with Juxtapp

<table>
<thead>
<tr>
<th>Feature Category</th>
<th>Examples</th>
<th>Our Approach</th>
<th>Juxtapp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dalvik bytecode opcode sequences</td>
<td>sget, goto, return</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Java VM type signatures</td>
<td>Z(Boolean), B(byte)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>String value of const-string instructions</td>
<td>-</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Type signatures for “invoked” functions</td>
<td>f(I,[B)Z</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

After extracting all the \( n \)-gram features, we then encode all the features in a bit-vector format fingerprint through a feature hashing technique using the \texttt{djb2} hash function. During the feature hashing process, we use a tuple \( A(i,j) \) to represent a feature position, in which \( i \) is the function offset indicating from which function the particular \( n \)-gram feature is extracted, and \( j \) is the bytecode offset indicating the position of the \( n \)-gram feature within the corresponding function. Then the feature-to-bit information is stored in a map, in which the key is the bit index within the fingerprint indicating where the feature is stored, and the value is the list of feature tuples that are mapped to the bit location. With increased feature space, we hope to reduce the number of the random feature collisions, and allow each bit index to represent the same \( n \)-gram feature content.

Similar to complete Android apps, individual legitimate libraries and the candidate malicious payloads are also represented in the same size of bit-vector fingerprints. The concrete \( n \)-gram size and the fingerprint size used for clustering are determined through analyzing the collision rate of random features, which is discussed in Section 3.5.2.

To measure the similarity between two fingerprints, we use the Jaccard index, or the Jaccard similarity, which is defined as the size of intersection divided by the size of union of two sets. Since each fingerprint is a bit-vector, we leverage cache-efficient bit-wise AND (\( \land \)) and bit-wise OR (\( \lor \)) operations to compute the intersection and the union. We use \( S(\cdot) \) to denote the number of 1-bits in the input, and define the similarity of two fingerprints \( fp_a \) and \( fp_b \) as follows:

\[
\text{Similarity}(fp_a, fp_b) = \frac{S(fp_a \land fp_b)}{S(fp_a \lor fp_b)},
\]

(3.1)
Our fixed-sized bit-vector fingerprint representation also allows us to easily measure the containment ratio in a similar fashion:

\[
\text{Containment}(\text{fp}_a, \text{fp}_b) = \frac{S(\text{fp}_a \land \text{fp}_b)}{S(\text{fp}_a)},
\]

which measures how much of the content of \(\text{fp}_a\) is contained in \(\text{fp}_b\).

### 3.2.2 Fingerprint Based Library Code Removal

To precisely remove legitimate library code without excluding a possibly injected malicious payload, we exclude legitimate library code from an app by removing the library-mapped bits from the app bit-vector fingerprint. For each legitimate library, we collect its official jar file and disassemble it into Dalvik bytecode sequences; then apply the same feature hashing technique to map the \(n\)-gram features of the library code into a bit-vector fingerprint \(\text{fp}_{\text{lib}}\). We then flip all the bits in the library fingerprint to get \(\overline{\text{fp}}_{\text{lib}}\). Since the same features contained in an Android app and the library are mapped to the same bit positions in their fingerprint representation, we can exclude library-mapped bits from an app fingerprint by bit-wise ANDing \(\overline{\text{fp}}_{\text{lib}}\) and \(\text{fp}_{\text{app}}\). Figure 3.3 demonstrates the overall procedure to safely remove legitimate \texttt{twitter4j} library code from a malware sample.

![Figure 3.3: Example procedure to safely remove legitimate “twitter4j” library code](image)

We first conduct statistical analysis for the disassembled apps to identify the embedded legitimate libraries, and record the years when the target samples were created. We then obtain\(^2\)

\(^2\)We randomly select one version of library in each year in case there are multiple versions of libraries released within the same year.
the officially released library jar files to create the corresponding library fingerprints, and remove
the library code from the analyzed apps. The library code removal process is applied only when an
app contains code snippets that are defined under corresponding library namespaces.

In our implementation, each library is represented with an individual fingerprint. We encode
multiple versions of the same library together in a single library fingerprint. This aggregated library
representation may cause a feature collision between the app code and the irrelevant versions of the
library code. However, we empirically demonstrate in Section 3.5.3 that the library code removal
process is precise because different versions of the same library typically share a high level of code
similarity due to code reuse, and the size of the single library is often smaller than the entire app.

3.2.3 Fingerprint Based Candidate Payload Extraction

The next operation is to extract malicious payloads from malware samples. We consider the
shared code segments (after excluding legitimate libraries) between each malware sample pair to
be a candidate malicious payload. For a group of malware samples, we obtain the intersection
of every fingerprint pair of library-excluded samples, and consider the shared 1-bits between the
sample fingerprints as a candidate payload fingerprint.

Figure 3.4 describes the intersection analysis procedure to extract a candidate malicious
payload at a high level. For two malware samples we first build their fingerprints and exclude the
legitimate library bits from the fingerprints. Then we pinpoint their shared 1-bits (e.g., bits index
2, 3, and 4) as potentially malicious\(^4\) bits and construct a candidate payload fingerprint.

\(^4\) Malicious payload mapped
During the candidate payload extraction process, we keep track of the association information between the candidate payload (e.g., A1-2) and the corresponding samples (e.g., A1 and A2). We subsequently use the payload-to-app association information and the malicious payload mining results to group malware samples.

3.2.4 Fingerprint Based Malicious Payload Reconstruction

Using the bit-vector fingerprint representation, we can also define the cluster fingerprint for a version of the candidate payload cluster as the intersection of all the candidate payload fingerprints in the cluster. The 1-bits contained in the resulting cluster fingerprint can be viewed as the shared malicious bits for all input apps that share the same version of malicious payload.

Using the identified malicious bits from app fingerprints, we can then reconstruct the corresponding malicious payload code by checking the feature-to-bit mapping information that was recorded during feature hashing, which can be viewed as the reverse procedure of fingerprint generation. Given the identified malicious bits, we locate the feature tuples that are mapped to those identified malicious bits. We use each retrieved feature tuple to locate the \( n \) lines of code where the \( n \)-gram feature is extracted, then reconstruct complete malicious code sequences by properly stitching the identified \( n \) lines of code segments together.

In practice, feature collision is possible but becomes negligible with appropriate \( n \)-gram size and fingerprint size, thus we will rarely recover irrelevant code. To a certain extent, payload code reconstruction compensates feature hashing collisions (e.g., resulting in missing \( n \)-grams) as far as the missing \( n \)-gram is within the overlapped original code sequences of recovered features. The reconstructed malicious payload code can be further inspected to verify its maliciousness.

3.3 Malicious Payload Mining

Based on our Android malware experiences, we obtained the following key insights: (a) In practice, when feature hashing is configured to have a low collision rate, malware app fingerprints will not contain a large number of shared 1-bits unless they do share certain common features (e.g., payload code snippets). (b) Likewise, if a target dataset contains malware samples that do share the same version of the malicious payload, then the candidate payload fingerprints extracted from those
samples will contain similar shared 1-bits and be automatically clustered into the same group. (c) After removing legitimate library code from an app, similar malicious payloads have higher chances to form a larger cluster than the ones related to less popular libraries or coincidentally shared code segments. (d) Compared to coincidentally shared code segments, similar malicious payloads will have a larger shared code base because of “legitimate” reasons for code reuse in the same malware family, and the fingerprints for the malicious payloads will have a larger amount of shared 1-bits.

Based on the above key insights, we designed the following strategies to iteratively select representative candidate payload clusters based on payload popularity, which is determined based on the three criteria: the entry size of a payload cluster $l$, the number of distinct apps associated with a payload cluster $m$, and 1-bits count of a payload cluster fingerprint $k$.

- We count the number of candidate payload fingerprint entries in each cluster, and maximize the possibility of extracting core malicious payloads by selecting the clusters with the largest number of payload fingerprint entries. Payload cluster size $l$ is a direct indicator for the popularity of the shared code segments between malware samples, and such popular shared code is a good candidate for one version of malicious payloads since we have already filtered out popular legitimate library code.

- We measure the distinct apps $m$ that contribute to generating candidate payload fingerprints of each cluster, and select the clusters with the largest number of distinct apps if they have the same number of payload entries. Payload clusters that contain a large number of unique payload entries are often associated with a large number of distinct apps, and we use this app association information to break the tie in case the number of cluster entries are the same since distinct apps can be considered as another sign of comparative popularity.

- We obtain the intersection bits $k$ of payload fingerprint entries in each cluster as the cluster fingerprint. If two clusters are associated with the same number of distinct apps, we then select the one with the larger number of 1-bits in its cluster fingerprint. In this way, we can extract the payload with a larger code size, and it helps to increase the likelihood of getting malicious
payloads together with shared libraries, and we subsequently exclude possibly remaining libraries later.

- During cluster selection, we keep track of which apps have been used to generate candidate payload fingerprints in the previously selected clusters, and consider already-selected apps as “inactive”. We update the remaining payload clusters by removing candidate fingerprint entries that are associated with “inactive” apps. Skipping such fingerprints allows us to extract one version of the malicious payload from each app. This helps to merge all the shared core malicious code together, and only extract the widely shared malicious code between all apps, which also helps to reduce the probability of extracting non-malicious payload code.

- We omit a payload cluster if the corresponding cluster fingerprint contains less than the minimum $k$ number of 1-bits, meaning that the extracted code segments are too small. It forces the algorithm to break the current large payload cluster into smaller clusters with a larger code size, and prevent different malware families from being clustered together. We set the minimum number of 1-bits $k$ to 70 since the majority of the analyzed Android malware app fingerprints had more than 70 1-bits.

- We exclude a candidate payload cluster if it becomes empty after the update in the last step, or if the number of payload fingerprint entries is too small (e.g., $l = 1$). This is because Clusters with only a single candidate payload entry provide little additional popularity information, and are more likely to contain less popular libraries or other coincidentally shared code snippets. We consider malware samples associated with such payload clusters as unclustered, and the unclustered app is evaluated as a singleton.

The shared payloads between Android samples can be library code segments, malicious payloads, copy-and-pasted code segments, or other coincidentally shared code segments. The above payload mining strategy enables us to select the most likely malicious candidate payload groups. Legitimate non-library reused code may be collected together with malicious payload only if it is shared across a significant number of apps. Otherwise, the less popular legitimate non-library code will be excluded during the (popularity-based) payload mining procedure. If the same benign app
is indeed used by many malware apps, we can further exclude original benign app code (i.e., the legitimate non-library reused code) in a similar way to remove library code using a benign app fingerprint.

3.4 Optimize Overall Clustering Efficiency

According to the previously discussed malicious payload mining procedure, we will generate $\frac{n \times (n-1)}{2}$ versions of candidate payload fingerprints given $n$ malware samples, but the hierarchical clustering algorithm also has a quadratic complexity with respect to the number of analyzed targets. Due to the overall quadratic complexity of the algorithm, directly using it to analyze large number of samples becomes a time-consuming task. Therefore, we further develop two methods to improve the scalability of the clustering analysis procedure, and hereafter refer them as Opt-1, and Opt-2.

3.4.1 Opt-1: Optimize Each Pairwise Computation

The first method to speed up the overall clustering process is to optimize each pairwise computation. Broder proposed minHash [103] to quickly estimate the Jaccard similarity of two sets without explicitly computing the intersection and the union of two sets. By considering our bit-vector fingerprint as a set, we apply minHash to further transform a large fingerprint into a smaller size signature, and calculate the similarity of minHash signatures to estimate the Jaccard similarity of the original fingerprints.

To apply minHash, we define a minHash function output of our bit-vector fingerprint $h(fp)$ to be the first non-zero bit index on a randomly permuted bits order of the fingerprint. We then apply the same minHash function to two fingerprint $fp_a$ and $fp_b$. This will generate the same minHash value when the first non-zero bit indexes for two fingerprints $fp_a$ and $fp_b$ are the same. Since the probability that the firstly encountered bit is a non-zero bit for $fp_a$ and $fp_b$ is conceptually the same as Jaccard similarity $\text{Similarity}(fp_a, fp_b)$ [104], we use the probability $\Pr[h(fp_a) = h(fp_b)]$ to estimate the original Jaccard similarity.

The probability estimation becomes more accurate if more independent minHash functions are used together. Formally, we define a minHash signature $\text{sig}(fp)$ to be a set of $k$ minHash function values extracted from $k$ rounds of random permutations over the fingerprint, and represent it as
follows: \( \text{sig}(\text{fp}) = [h_1(\text{fp}), h_2(\text{fp}), ..., h_k(\text{fp})] \). We denote the similarity of two minHash signatures as the ratio of equal elements between \( \text{sig}(\text{fp}_a) \) and \( \text{sig}(\text{fp}_b) \).

Instead of maintaining \( k \) random permutations over the bit-vector, we follow a common practice for using the minHash technique and use \( k \) different hash functions to simulate \( k \) random permutations, where each hash function maps a bit index to a value. In order to create \( k \) hash functions, we first generate \( k \) random numbers, then use the FNV [105] hash algorithm to produce a basic hash output for each bit index, and finally apply the XOR operation between each random number and the hash output to get the \( k \) hash outputs. For each hash function, we select the smallest hash value (to simulate the first non-zero bit index) over all of the bit indexes of the fingerprint as the final hash output.

Note that the FNV hash value and the \( k \) random numbers are all 32 bits unsigned integers, and they can be used to safely simulate random permutation over 512MB bit-vector fingerprints. In practice, the \( k \) value usually needs to be larger than 100 to generate good enough results [104]. We set \( k \) to be 256 in our experiments, and thus convert each bit-vector fingerprint into a 1KB minHash signature.

In order to evaluate the potential impact of Opt-1 on accuracy, we conducted two experiments on the smallest 50 malware families\(^4\): one experiment (Exp-1) with no optimization, and another experiment (Exp-2) using Opt-1. We used the clustering output from Exp-1 as a reference, and measured the precision and recall of the clustering output from Exp-2. The precision and recall indicate how similar the two experiments results are, and are used to check the impact on accuracy brought by Opt-1.

Our experiments showed that on average Exp-2 took less than 83\% time to complete compared to Exp-1 for the analyzed families, and the average precision and recall of the clustering output were 0.993 and 0.986. Opt-1 significantly reduced the overall memory consumption with minHash signature representation and improve the pairwise computation efficiency with almost zero accuracy penalty.

\(^4\)We selected those families since their maximum family size is under 100 and all the experiments for those families can be finished within 1 hour.
3.4.2 Opt-2: Employ Approximate Clustering

The previous optimization is still not sufficient for using the algorithm to analyze large scale malware samples. For instance, when analyzing with 2,000 samples, the algorithm will create 1,999,000 candidate payloads, and it results in approximately $2.0 \times 10^{12}$ pairwise comparisons. Even 1% of the total comparison still takes lots of computational resources. To resolve the scalability issue for a large dataset input, we further adopt a prototype-based clustering technique [97,106] to achieve approximate clustering.

Specifically, we randomly divide the target samples into small size (e.g., 150) groups. For each group, we apply hierarchical clustering analysis on the shared payload within the group, and create a prototype fingerprint for each payload cluster by applying intersection analysis (to obtain all the shared 1-bit) among the payload fingerprints in each cluster. We then conduct hierarchical clustering analysis on all the collected prototype fingerprints. In this way, we represent a group of similar payload fingerprints with a single prototype fingerprint, and the algorithm proceeds with approximate clustering analysis using the prototype fingerprints instead of the original payload fingerprints.

We design two experiments to evaluate the impact of Opt-2 on accuracy: one experiment (Exp-3) using Opt-1 only, and another experiment (Exp-4) using Opt-1 and Opt-2. Due to the quadratic complexity of the original algorithm, the overall analysis (using Opt-1 only) will get dramatically slower for analyzing a larger number of malware samples. For instance, we found it takes about one day to analyze 1000 samples and more than five days to analyze 2000 samples for Exp-3. In order to conduct the evaluation within a reasonable amount of time, we randomly selected 70% of labeled samples from the largest 4 malware families and conducted the two experiments for each family. We used the clustering output generated by Exp-3 as reference, and measured the precision and recall of the clustering output generated by Exp-4 to evaluate the accuracy impact brought by Opt-2.

Our experiments showed that on average Exp-4 can speed up more than 95% compared to Exp-3, and the average precision and recall for the analyzed 4 families were 0.955 and 0.932. This
optimization makes it feasible to apply our algorithm to analyze a bigger scale of malware families while providing a desirable trade-off option between speed and accuracy.

3.5 Experiments

We first describe the data preparation procedure, then report malware clustering results and key findings of our experiments.

3.5.1 Data Preparation

We obtained a large collection of potentially malicious Android apps (ranging from late 2010 to early 2016) from various sources, include Google Play, VirusShare [58] and third party security companies. In order to prepare ground-truth family labeling for the datasets, we queried the collected apps against VirusTotal [2] around April 2016, and used the scanning results to filter out potentially ambiguous apps.

To assign family labels to the collected malware samples, we applied the following steps: (1) tokenized VirusTotal scanning results and normalized the contained keywords, and then counted the total number of occurrences of each keyword. (2) removed all the generic keywords such as Virus, Trojan, and Malicious. (3) detected keyword aliases by calculating the edit distances between keywords. For example, Nickyspy, Nickspy, Nicky, and Nickibot were all consolidated into Nickispy. (4) assigned the dominant keyword as the family label for the sample. A keyword was considered as dominant if it satisfied two conditions: (a) the count of the keyword was larger than a predefined threshold $t$ (e.g., $t=10$), and (b) the count of the most popular keyword was at least twice larger than the counts of any other keywords.

Although our malware labeling process may look similar to AVclass [107], we developed the approach independently without the knowledge of the AVclass; and both works were finished around the same time. The unlabeled samples were not included in the malware dataset for clustering analysis. In summary, we collected 19,725 labeled malware samples from 68 different families, and the detailed breakdown of the malware samples is shown in Table 3.2.

Besides the above labeled malware dataset, we also collected Android Genome malware samples [30] to obtain an optimal clustering threshold, and randomly selected a list of 10,000 benign
samples from AndroZoo [108] to evaluate the accuracy of the library removal procedure. In particular, we selected benign apps that were created around the same time (before Jan 1st, 2016) as most of the labeled malware samples, and their latest (Mar 2017) VirusTotal re-scanning results showed no malicious labels.

3.5.2 Feature Collision Analysis

The accuracy of the proposed clustering system and the correctness of the reconstructed malicious payloads relies on the assumption that unique features will be mapped to unique bit locations within the bit-vector fingerprint. Feature collisions are directly impacted by two parameters: an $n$-gram size, and a bit-vector fingerprint size. To evaluate a feature collision rate, we varied the $n$-gram size (2 and 4) and the bit-vector fingerprint size, and then measured how many unique features were mapped to the same single bit position, i.e., feature collision. Figure 3.5 illustrates feature collisions with regard to different $n$-gram sizes and fingerprint sizes.

The graph shows that feature collisions occur more frequently when the fingerprint size is small. The total number of unique features depends on the $n$-gram size. For the labeled malware, it was about 4.1 million for 2-gram features, and 14.4 million for 4-gram features. And for the benign dataset, it was about 15.2 million for 2-gram features, and 45.3 million for 4-gram features. According to the *pigeonhole principle*, when putting $N$ unique features into $M$ buckets, with $N > M$, at least one bucket would contain more than one unique feature. This means that we need to
set the bit-vector fingerprint size larger than the total number of unique features to reduce feature collision. Therefore, we set the default $n$-gram size to be 2 and default fingerprint size to be 1024KB which provides 8,388,608 unique bit positions. With the above configuration, the unique feature per bit value was reduced to 0.49 to process the labeled malware dataset. Notice that the complete feature space is unlimited for our system due to the inclusion of arbitrary string values, however the true unique features contained in a certain dataset will be limited.

![Graph](image)

**Figure 3.5: Random feature collision status**

**Figure 3.6: Benign apps lib removal accuracy**

### 3.5.3 Library Removal Accuracy

Besides the random feature collision discussed in the previous section, it is also possible that feature collision may happen between the app code and the irrelevant versions of the library code. To evaluate the library removal accuracy, we assumed the libraries used in benign samples were not purposefully manipulated, and measured the precision (e.g., how much of the removed code is true library code) and recall (e.g., how much of the true library code is removed) of library code removal results for the prepared benign samples. Particularly, we considered the code that were defined under the official library names in the benign samples as ground truth library code, and created the *true library code* fingerprint $f_{\text{true}}$ by mapping all the features from the true library code to a bit-vector fingerprint. After removing the library code from each app, we identified the bit positions that were presented in the original app fingerprint and were removed subsequently;
and used the identified bit positions to generate removed library code fingerprint $\text{fp}_{\text{removed}}$. Using the containment ratio calculation function as discussed in Section 3.2.1, library removal precision $P_{\text{lib}}$ is defined as $\frac{S(\text{fp}_{\text{true}} \land \text{fp}_{\text{removed}})}{S(\text{fp}_{\text{removed}})}$, and library removal recall $R_{\text{lib}}$ is defined as $\frac{S(\text{fp}_{\text{true}} \land \text{fp}_{\text{removed}})}{S(\text{fp}_{\text{true}})}$, where $S(\cdot)$ denotes the number of 1-bits in the bit-vector.

Figure 3.6 depicts the library removal precision and recall for the benign apps. We observed that 9,215 benign apps contained at least one legitimate library, and the median values for precision and recall were 0.94, 0.95, respectively. We manually inspected certain corner cases with poor precision or recall. The poor precision cases were due to incomplete true library code extraction, e.g., an older version of Admob library contained obfuscated versions of code which were not under the com.google domain, and are not counted as true library code. The poor recall cases were due to excessive true library code inclusion, e.g., all the code of the Androidify app was defined under the com.google domain which made the distinction of library code obscure.

3.5.4 Malware Clustering Results

In order to select an optimal clustering threshold for the system and assess the performance compared with other known Android malware clustering systems, we first applied our clustering system on the Android Genome malware dataset. We used the classical precision and recall [8, 11, 42, 96–98] measurements to evaluate the accuracy of clustering results. Figure 3.7 describes the clustering precision and recall results with various thresholds.

The highest F-measure score was 0.82 with precision of 0.90 and recall of 0.75 when the clustering threshold was 0.85. We set the default clustering threshold value to be 0.85 for subsequent clustering analysis. As a reference, ClusTheDroid [28] achieved precision of 0.74 and recall of 0.73 while clustering 939 of Android Genome malware samples.

Note that the clustering outputs produced by our system is per sub-version instead of per family, therefore it is more challenging to properly obtain fine-grained ground truth labels to evaluate the accuracy. In fact, this was the main reason for the slightly low recall of our system with respect to coarse-grained ground truth labels, e.g., one Android malware family’s samples might contain multiple versions of malicious payloads. While reviewing the clustering results, we noticed
that 13 families of the Genome dataset contained more than one versions of malicious payloads. For example, Basebridge contained 7 versions of malicious payloads with a threshold of 0.85.

Therefore, we separated the labeled malware samples into sub-versions using the clustering system, and further designed several experiments to evaluate the clustering results with a manually verified sub-version ground-truth. We manually verified the correctness of the sub-version cluster results. For the generated sub-version clusters, we first checked if the extracted payload was the indeed malicious. Since each version of the extracted payloads usually had similar class names and Dalvik code sequences, the maliciousness of the extracted payload can be spotted by checking
the extracted class names (e.g., similar pseudo-random pattern). In case the class names were not enough to determine its maliciousness, we then went through the reconstructed code segments and checked if there were any suspicious activities or behaviors, such as stealthily sending out premium SMS. After verifying the maliciousness of the extracted payload, we then randomly selected 3 samples from each sub-version group, and checked if the selected apps contained the same version malicious payload. Out of 19,725 malware samples that were labeled with 68 families, we obtained a total of 260 verified sub-version clusters, and each cluster corresponded to one version of the malicious payloads.

We considered the VirusTotal family labels together with the manually verified sub-version information as ground truth, and prepared 10 datasets for experiments. For each dataset, we randomly selected 30 sub-versions from the entire ground truth dataset (e.g., 260 sub-versions), then mixed the selected samples together to create dataset. The resulting datasets had different overall sizes as each individual sub-version had different numbers of samples. The detailed dataset sizes and sample clustering results for the 10 datasets are presented in Table 3.3. On average, the sample clustering algorithm separated the input malware samples into 29.9 clusters, which was extremely close to the reference set (i.e., 30 sub-versions). For the 10 datasets, the clustering algorithm achieved average precision of 0.984 and average recall of 0.959, the worst precision and recall for clustering multiple malware families were 0.971 and 0.858, which suggests that the clustering system generated consistent and reliable outputs.

3.5.5 Key Findings for Malicious Payload Analysis

The key findings learned from the malware sub-version verification process are listed as follows.

- **Significant library code ratio**: From the labeled malware datasets, we found that the average library code ratio was larger than 50% for the malware samples that contained at least one legitimate library. This highlights that existing Android malware similarity analysis work becomes ineffective without properly handling library code.
• **Limited versions of malicious payloads:** During our experiments, we acquired 260 versions of malicious payloads from 68 labeled malware families while conducting clustering of each family. Among the 68 malware families, 27 families had only one version of malicious payload, and 5 families had more than 10 different versions of malicious payloads. For example, **Dowgin** was the largest malware family and had 23 versions of malicious payloads extracted.

• **Malicious payload under popular namespaces:** We conducted manual analysis on the extracted malicious payloads, and noted that 29% of Android malware families injected their malicious payloads under popular namespaces, such as **com.google** and **com.android**, or legitimate advertisement library namespaces like **com.umeng**. Table 3.4 includes the detailed malicious payload findings for the identified families. Since **com.google** and **com.android** are the main class names used by Android Open Source Project and Google Mobile Services, such malicious payloads can easily get overlooked.

### 3.6 Limitations

Our Android malware clustering approach is based on the assumption that malware authors often reuse the same malicious payload to create new malicious samples, and the obfuscated code sequences of malicious payload would largely remain the same if they are generated by the same obfuscation tool. This is consistent with our findings as listed in Section 3.5.5. Theoretically, advanced obfuscation techniques (e.g., class encryption or dynamic loading) can eventually break the assumption by generating a new version of a malicious payload for every new malware instance, or completely removing the original malicious payload from **classes.dex**. The attack and defense against malware obfuscation is a long-term arms race, and as already been observed in the traditional desktop malware analysis domain. For example, as observed in desktop malware research [93–95], independent systems might be desirable to specifically handle the de-obfuscation process. We consider it as a separate pre-processing step for malware analysis, and consider a comprehensive solution for advanced obfuscation an orthogonal problem. In addition, using dynamic analysis with a sandbox can help further analyze malware. However, dynamic analysis also suffers from its own limitations, such as sandbox evasion and code coverage.
<table>
<thead>
<tr>
<th>Family</th>
<th>Popular Class Names Used</th>
</tr>
</thead>
</table>
| Nickispy | com.google.android.info.SmsInfo  
com.google.android.service.UploadService |
| Uuserv | com.uuservice.status.SysCaller.callSilentInstall  
com.uuservice.status.SilenceTool.MyThread.run |
| Fjcon | com.android.XWLauncher.CustomShirtcutActivity  
com.android.XWLauncher.InstallShortcutReceiver |
| Yzhc | com.android.Base.Tools.replace_name  
| Gumen | com.umeng.adutils.adsConnect  
com.umeng.adutils.SplashActivity |
| Basebridge | com.android.sf.dna.Collection  
com.android.battery.a.pa |
| Spambot | com.android.providers.message.SMSObserver  
com.android.providers.message.Utils.sendSms |
| Moavt | com.android.MJSrceen.Activity.BigImageActivity  
com.android.service.MouaService.InitSms |
| Zitmo | com.android.security.SecurityService.onStart  
com.android.smon.SecurityReceiver.sendSMS |
| Mseg | com.google.vending.CmdReceiver  
android.ad.appoffer.Copy_2_of.DownloadManager |
| Droidkungfu | com.google.ssearch.SearchService  
com.google.update.UpdateService |
| Dowgin | com.android.quishui.app.dmc  
com.android.game.xiaqiang.jokes.Data9 |
| Fakeinst | com.googleapi.cover.Actor  
com.android.shine.MainActivity.proglayss.Click |
| Ksapp | com.google.ads.analytics.Googleplay  
com.google.ads.analytics.ZipDecryptInputStream |
| Bankun | com.google.game.store.bean.MyConfig.getMsg  
com.google.dubest.eight.isAvilible |
| Pjapps | com.android.MainService.SMSReceiver  
com.android.main.TANCActivity |
| Adwo | com.android.mnreader1030  
com.google.ads.AdRequest.isTestDevice |
| Svpeng | com.adobe.flashplayer_FWD.oInBackground  
com.adobe.flashplayer_FA.startService |
| Opfake | com.android.appupdate.UpdateService  
com.android.system.SurpriseService |
| Badao | com.google.android.gmses.MyApp  
com.android.sechone.FileUtil.clearTxt |
We believe that the Android malware analysis community can benefit from our work in several aspects. (a) It offers an alternative malicious payload extraction approach in which we can extract a more complete version of malicious payloads even if the malicious payloads are injected under popular library names or under existing functions. (b) It provides a viable solution for conducting Android malware clustering analysis by checking if malware samples contain the same version of malicious payloads. (c) Significant number of Android malware samples are not obfuscated or obfuscated by simple obfuscation tools, even for the samples we collected recently. For example, within the extracted 260 versions of malicious payloads, we observed 181 of them had plain code, and only 79 of them used naming obfuscation, which was a simple basic obfuscation technique. (d) As long as there are shared malicious code segments regardless of obfuscation among the samples from the same malware family, our algorithm extracts the shared patterns and uses them for deciding malware clustering output.

3.7 Summary

In this chapter, we proposed a practical solution to conduct Android malware clustering. As an internal component, the fingerprint based library removal technique was used to distinguish a legitimate library and a bogus library that may share the same library name. Unlike traditional clustering techniques which examine the overall similarity, we achieved Android malware clustering by checking whether the analyzed Android malware samples shared the same version of malicious payload code. Compared with existing malicious payload extraction systems, our approach extracts malicious payloads even if they were injected under popular library namespaces or under existing benign functions, and it provides a more complete picture of the whole malicious payload. Our comprehensive experimental results demonstrate that our clustering approach generates consistent and reliable outputs with high precision and recall.
CHAPTER 4: EVALUATION OF MACHINE LEARNING BASED MALWARE DETECTION SYSTEMS WITH UNKNOWN KNOWLEDGE

The number of available Android applications has continued to grow in recent years, reaching about 3.6 million apps in the Google App Store in February 2018 [109]. With their huge market share, Android devices become popular targets for attackers. Malicious authors can use sophisticated techniques to evade existing vetting systems, and the undetected malicious apps (especially within official app stores) have become a major threat to end users. Correct and timely vetting of all submitted apps at a large scale has become demanding, given the sheer volume of apps that are already in the markets and the thousands of new apps entering markets on a daily basis [110].

Much previous work [40, 41, 44, 46, 48, 111] has demonstrated that machine learning (ML) techniques can help the triage process for Android malware detection. To create an effective ML classifier, researchers need to train the ML classifier with a ground truth dataset. And to examine the effectiveness of such ML systems, an ML classifier needs to be evaluated on a dataset with known ground truth.

There is a critical problem, though, in existing ML system evaluation approaches in this application domain. Based on the best of our knowledge, most machine learning based Android malware research obtains the ground truth by consulting AV products, and uses the same label set for training and testing (albeit on disjoint subsets). We argue that the results from such evaluations offer little insights into the utility of the ML approach in practice. The expected use of an ML-based malware classifier in practice is to triage apps that are yet to be flagged by any AV products. For example, a security company may receive a large number of suspicious app files on a daily basis, which have not been known by any of the AV products to be malicious. An ML-based approach can be used to predict the likelihood the apps may be malicious, so precious human analyst time
can be prioritized to the more likely malware samples. In this application setting, the utility of an ML-based classifier lies in whether it can accurately predict “zero-day” malware – malware that has yet to be flagged by any AV products. Indeed, this is probably true for any meaningful application of machine learning in malware detection. If malware is already flagged by AV products, there is not much use for an ML classifier to tell us the same thing. To evaluate the effectiveness of an ML approach to malware triaging, we shall test the classifier on samples that are not yet known to be malware at the time the classifier is trained.

Specifically, for the problem of Android malware detection, if a classifier is trained based on some existing knowledge, its utility can only be demonstrated by how well it can predict that an app, not yet known by the existing knowledge to be malware, is actually malware. To this end, we consider a malicious app as zero-day malware if no AV products can detect it at a specific time, and propose to evaluate machine learning outputs by checking zero-day malware detection instead of known malware detection results. We designed two separate experiments to demonstrate ML approaches can be used for zero-day malware detection.

In this section, we propose to evaluate machine learning outputs by checking zero-day malware detection. Real world malware detection is very time-sensitive and heavily relies on knowledge about the malware family. Observing the mismatch between how ML systems have been evaluated in previous malware detection work and the true purpose of malware detection in practice, we argue that the usefulness of a real world machine learning system is not in verifying or reproducing what has already been known (e.g., confirmed malware). Rather, it is most helpful if it can be used to produce verifiable results that are previously unknown (e.g., detect zero-day malware), which arguably is the very goal of using machine learning in this problem domain.

4.1 Issues with Traditional ML Based Malware Detection

Traditional machine learning techniques have been frequently used for malware detection, in which the training and testing datasets are completely separate from each other, but prepared using the same ground truth knowledge. And previous ML based malware detection systems mainly evaluate the effectiveness of the designed system by checking the detection results with regard to AV scanning results at a specific time. Particularly, given a training dataset $T$, a traditional machine
learning approach first learns a classifier $C$ on $T$. Then it uses $C$ to predict the class labels for the samples in a disjoint testing dataset $S$. The prediction (i.e., detection result) is compared with the ground truth of the samples in $S$, to evaluate its effectiveness. Commonly, a detected sample is considered as true positive if it is also detected (i.e., confirmed) by a certain number of AV products, and false positive if it is not detected (i.e., unverified) by any AV products.

However, we observe that evaluating with the known malware creates a dilemma for demonstrating the practical utility of the ML based malware detection system. On one hand, the correctness of the malware detection results needs to be verified by checking against the ground truth references. On the other hand, if the testing apps are known malware (i.e., detectable by existing antivirus products), such evaluation only shows that the proposed malware detection system can reproduce the same detection capabilities of existing antivirus products, instead of demonstrating detection capabilities superior to existing antivirus products.

We also note that the traditional evaluation strategy provides no insights when comparing the proposed ML based malware detection system against any individual existing antivirus product. To prepare a large set of ground truth datasets, researchers often use the majority voting information of a collection of antivirus scanning results for the target dataset. Using such an ensemble created dataset for training purposes means the proposed ML system already utilizes certain intrinsic malware detection knowledge derived from existing antivirus products, thus it will be no surprise to see that the trained ML model has better detection results than any individual product.

In addition, the benign and malicious labels for the training dataset are created using existing AV scanning results, which may be highly unreliable [107,112], and heavily depend on AV products' detection capabilities at a specific time. In a real world situation, it is challenging to completely eliminate label noise from a training dataset, especially for the benign labels in the training dataset if zero-day malware exists. For example, we have observed that over a period of time, some apps' scanning results were changed from benign to malicious. Such mislabels in the data may impact the quality of the classifier trained from it, and also produce misleading results when doing cross-validation testing.
Even though traditional ML based ML detection systems have those issues, we still frequently observe new ML based malware detection systems claiming impressive results during their evaluation. But the detection accuracy in the real world may never be the same as those obtained in their controlled experiments. The first step to bridge the gap between experimental accuracy and real-world accuracy is to recognize the reality that ML based systems will generate outputs with 100% certainty which means false positives/negatives will be always inevitable; The second step is to assign proper role ML based malware detection systems and adjust expectations for using ML based malware detection systems in practice: (1) ML based malware detection systems are not suitable for making the final maliciousness decision about the target sample due to their intrinsic lack of specificity; For example, it is useless for end users when the ML based malware detection system reporting target sample as “malicious (confidence 75%)”, since users will need to resort to other resources to make a final decision. (2) Nevertheless, outputs generated by ML based malware detection systems are still helpful for security companies in that they provide strong hints about target sample’s potential maliciousness which means the ML based malware detection systems are perfect for data triaging procedure.

Given the above reflections, we believe it will generate more meaningful results if the ML based malware detection systems are evaluated in a similar manner that embodies their best strengths and admits their inherent weaknesses at the same time. Therefore, we propose to evaluate machine learning outputs by checking zero-day detection capabilities instead of inspecting the detection accuracy with those already confirmed malware. We want to emphasize that the focus of this work is not concrete evaluation results (e.g., zero-days) but the overall evaluation process, i.e., we are not designing yet another ML based zero-day Android malware detection system.

4.2 Zero-day Malware Detection Based Evaluation

In this section, we describe the machine learning approaches that lead to zero-day malware detection in practice, and the evaluation strategy for verifying zero-day malware detection results. A malware app is a zero-day malware if no AV products can detect it at a specific time \( t \). A more well-known definition of zero-day malware is that no AV products can detect it at the current time. Once the malware has been studied, it becomes known to the public an malicious and the malware
shouldn’t be considered zero-day anymore. We describe the machine learning approaches that lead to zero-day malware detection in practice, and an evaluation strategy for verifying zero-day malware detection results.

4.2.1 Malware Prediction Approach

Existing ML based malware detection systems [40, 41, 44, 46, 48, 111] are commonly referred to as malware classification or malware prediction. During the malware prediction process, an ML model is built upon the provided ground truth training dataset, and the ML model is then used to analyze the target samples and generate a prediction for each sample indicating its maliciousness.

![Diagram of malware prediction experiment overview]

In addition to the well-known ML algorithms, we describe a special algorithm called Label Regularized Logistic Regression (LR-LR) which can be used for malware prediction scenario. The LR-LR approach is a semi-supervised variant of Logistic Regression, training over only positive and unlabeled data. It was recently applied to Android malware detection as a means for potentially detecting inaccuracies in existing VirusTotal-labeled ground truth [113]. Instead of using the benign class as negative examples and attempting to learn the difference between positive and negative classes, LR-LR trains upon the positive data, and then regularizes the class distribution in the unlabeled data towards an expected distribution. Specifically, an expert-provided $\tilde{p}$ constant is used, which represents the expected proportion of positives in the unlabeled dataset. The optimization function then regularizes the current expectation of positives in the unlabeled class into what it
should be, as defined by $\tilde{p}$. Eventually, a classifier that can predict positives and negatives is trained only upon positive data, unlabeled data, and an expectation of how many positives exist in the unlabeled set. The novelty of the approach is that it allows utilization of benign data as unlabeled data, thus avoiding relying upon benign labels which are inevitably noisy. Still, the LR-LR approach leverages elements of the data that can result in a better classifier.

We show the experiment overview of using malware prediction techniques for zero-day malware detection in Figure 4.1. The major characteristic for the malware prediction approach is that the ML model is trained and applied on disjoint datasets. Since the main focus of this work is zero-day malware detection, we do not include any known malicious samples in the testing dataset. In order to show verifiable zero-day detection results, we prepare two sets of ground truth labels for the testing dataset, and use the first set of ground truth labels to get the ML reported zero-day detection results, and use the second ground truth labels to verify the zero-day detection results. We describe the label upgrading techniques in Section 4.2.3.

4.2.2 Mislabel Identification Approach

In practice, we can create a cleaner malicious training dataset by tuning the maliciousness decision threshold. For example, only consider the samples that are detected as malicious by at least $n$ different AV products, and exclude the rest for training purposes. But for the benign training dataset, there is only one way to prepare the training dataset: consider all the samples that are not detected as malicious by any AV products to be benign. This may result in poor performing ML models if the benign training dataset contains noise (e.g., the zero-day samples). Therefore, we propose to detect malware using mislabel identification techniques, which are designed to identify the incorrectly labeled samples in a dataset. Intuitively, if we consider all the target samples in the testing set as benign, the malware detection objective can be viewed as identifying mislabels in the benign dataset. Compared with traditional approaches, mislabel identification approaches directly work with the target dataset, thus eliminating the requirement for preparing a reliable benign training dataset.

We describe a special mislabel identification approach named Active Label Noise Removal (ALNR) which has been proposed recently. ALNR is a two-stage process to identify the label
noise from the training dataset. This approach is built on top of the SVM [114] algorithm and designed [115] based on the observation that the maximum margin principle used in SVM has the characteristic of capturing the mislabeled samples as support vectors; thus the label noise examples contained in the training dataset will likely be selected as support vectors for the trained SVM classifier. In \textit{Stage-1}, ALNR uses all the provided malware and “benign” samples to build a standard SVM classifier $A$ and finds the support vectors for the provided dataset. In \textit{Stage-2}, it removes the support vectors (SVs) and uses all the remaining non-SV samples to build a new classifier $B$. Then, it uses classifier $B$ to classify the benign SVs identified from Stage-1. If $B$'s predicted labels are not the same as the original provided benign labels, the algorithm reports the sample as a potential mislabel in the benign dataset.

In addition to special mislabel identification algorithms like ALNR, we can also use the traditional ML approaches to achieve mislabel identification by directly training and testing on the same dataset, which shows the bias of the classifier and what the classifier can not learn from the training dataset since some of these examples may be mislabeled.

![Figure 4.2: Mislabel identification experiment overview](image)

We show the experiment overview of using mislabel identification techniques for zero-day malware detection in Figure 4.2. The major characteristic for the mislabel identification approach is that the ML model is trained and applied on the same datasets. However, since the malicious training dataset is prepared with high quality (\textit{i.e.}, large AV detection threshold), we only use the
benign training dataset for testing purpose. In order to demonstrate verifiable zero-day detection results, we have to prepare two sets of ground truth labels, using the first version of the ground truth labels to get the potential zero-day detection results, and using the second version of the ground truth labels to evaluate the zero-day detection results. We describe the label upgrading approaches in next Section.

4.2.3 Evaluation of Zero-day Detection Results

Since the real ground truth of an app’s maliciousness is not readily available for zero-day malware, it is challenging to evaluate the zero-day malware detection results. We assert that the key step to enable verifiable zero-day malware detection is to prepare two sets of ground truth labels for the testing datasets. In particular, we use the first set of ground truth information to identify the potential zero-day detection results, and apply the second set of ground truth labels to determine the real zero-day detection results. We present two ways to prepare different ground truth sets and evaluate the effectiveness of zero-day malware detection results.

The first way to prepare two sets of ground truth is to scan the same datasets multiple times. For example, given a large number of Android apps, we can scan them against VirusTotal at time $t_1$ to get the first set of ground truth information, and scan them at time $t_1 + \delta$ to get the second set of ground truth information for the dataset. The intuition is that if we wait a period of time and re-scan the app samples, we may find that some apps’ scanning results change from benign to malicious due to AV vendors’ updated malware detection capabilities, indicating that the app was a zero-day malware earlier but later discovered as malicious by the AV vendors. Thus, examining historical data on AV scanning reports can help retrospectively examine whether an ML-based system can identify zero-day malware at an earlier time.

The second way to achieve verifiable zero-day malware detection is to use one set of ground truth information (e.g., AV scanning results) to filter the potential zero-day detection, then manually inspect the detection result to identify the real zero-day detection results. When using an ML system for real world malware detection, users typically want to train with best available ground truth, and the same ground truth information can be used to identify the potential zero-day results. We believe this process reflects a more useful way of using ML based malware detection in practice:
ML based malware detection has long been criticized for the inevitable false positive detection results, which means further inspection of the ML detection results is usually needed; combining the ML detection results and existing AV detection capabilities helps to prioritize the really interesting ones (e.g., zero-day malicious apps).

4.3 Experiment Preparation

We collected Android apps from various sources, such as PlayDrone [116], AndroZoo [108], VirusShare [58], internet service providers, and security companies. The apps are separated into the following three major categories: scanned malicious samples, scanned benign samples, and unscanned samples.

4.3.1 Malware Samples

We used AV scanning results to prepare the malware dataset. To obtain AV scanning results, we query the MD5 of the samples against VirusTotal, which is an online service that integrates more than 50 different AV scanners. Particularly, we consider a sample detected by at least \( n \) number of different AV products as confirmed malicious, a sample detected by less than \( n \) AV products as unconfirmed.

In this work, we decided to use the AV detection threshold of 10 to prepare the malicious training dataset, since the scanning results for such malware samples are more stable. We first used the AV scanning reports obtained in April 2016 to prepare the malware dataset, and randomly selected 15,000 malicious samples from 188,355 high quality malicious samples. We didn’t use all available malicious samples because: (1) it will take a significantly longer time to conduct the experiment using all available malicious samples. (2) previous research [117] showed that the malware to benign class ratio of the training dataset could impact the performance of the trained classifier. (3) we decided to configure all of our experiments with a tractable class ratio (malware vs benign) of 1:10, to consider the unbalanced ratio of malware and benign apps in reality and avoid the bias towards an overwhelming amount of benign samples.

Note that to report the final zero-day detection results, we also checked the VirusTotal scanning results for the corresponding samples. We used AV detection thresholds \( n \) of 2, 5 and 10 as
the maliciousness decision criteria, since the samples that are confirmed by only a few AV products may not be very reliable.

### 4.3.2 Benign Samples

A scanned benign sample means that the MD5 of the app was queried against VirusTotal, and no malicious flags were contained in its scanning report at the querying time. However, since scanning data for a particular app may be outdated, the existing scanning results may not represent the AV products' latest malware detection capabilities. Therefore, we conducted multiple rounds of scanning within approximately two years to obtain the upgraded ground truth labels for the same dataset.

We used the scanning reports obtained before April 2016 as the resources for preparing our first benign sample dataset, and removed those samples that hadn’t been analyzed by VirusTotal by April 2016. That is, all of the scan dates contained in those reports are earlier than April 2016, and none of the AV products had detected them as malicious as of April 2016. In the end, 811,649 Android samples were labeled as benign since no malicious label was returned for those samples.

For the same dataset, we again collected their scanning reports around December 2017. We observed that the majority of the scan dates for those samples were updated in 2017, which means they were rescanned in 2017. According to the newly obtained scanning reports, we observed 751,804 of them were still detected as benign samples. Even though a substantial amount of samples were changed from benign to malicious, we noticed that about 69% of them are only detected by one AV vendor, which means the newly assigned malicious labels maybe contain significant false positives.

### 4.3.3 Unknown Samples

We also selected unknown Android samples which were not queried against VirusTotal before. Particularly, we checked the timestamp of the main Dex file contained in each unscanned Android app, and only selected those whose timestamps were later than June 30, 2016. Thus, all of the unknown samples were newer Android apps compared with the labeled malware samples and benign samples. In the end, we obtained 413,353 unknown Android samples.
Table 4.1: Summary of the prepared datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Data labeling resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware M-2016</td>
<td>15000</td>
<td>Apr 2016 scan results</td>
</tr>
<tr>
<td>Benign B-2016</td>
<td>811649</td>
<td>Apr 2016 scan results</td>
</tr>
<tr>
<td>Unknown U</td>
<td>413317</td>
<td>-</td>
</tr>
</tbody>
</table>

To sum up, we present the overall dataset in Table 4.1. For the rest of the chapter, we refer to the 15,000 malware samples obtained using April 2016 AV scanning results as malware dataset M-2016, the 811,649 benign samples obtained using April 2016 AV scanning results as benign dataset B-2016, and the unknown samples (as of June 30, 2016) as unknown dataset U.

4.3.4 Feature Construction

Since the focus of this work is not about feature construction, but to provide a new approach for using existing machine learning algorithms to effectively detect real world zero-day malware, we decided to use previously known successful Android malware features.

Particularly, we used the same set of Android app features as defined in [117], because the overall feature set is relatively small, e.g., compared with Drebin [48] which may end up with millions of unique features. Further, it contains the majority of Android app features that have been shown to work well in previous Android malware research [45,47,52].

To obtain the features, we conducted lightweight static analysis of the Android apps, and extracted 471 different features which can be separated into the following categories: critical API usage, permission requests, intent action, obfuscation characteristics, native code signatures, etc. Note that the values for all of the features are limited to 1 and 0, which is used to indicate whether the app contains the particular feature or not. Since the features are already condensed, we do not further apply any feature selection process, i.e., all subsequent experiments are conducted with the same set of feature vectors.

4.4 Malware Prediction Experiment

We design an experiment to achieve verifiable zero-day malware detection using malware prediction techniques. Verifiable means the samples are initially identified as benign and later
confirmed as malicious, thus showing that the samples are real zero-day malware at the time when they are detected by the ML models.

In this experiment, we used one popular traditional machine learning algorithm—Random Forests (RF)—as the first representative traditional machine learning approach, since it is efficient and can usually generate good results without much tuning effort; and used the Label Regularized Logistic Regression (LR-LR) approach as the second malware prediction approach since its ML model does not rely on a benign training dataset. Because mislabel identification techniques can be used to remove mislabels, we also used the Active Label Noise Removal (ALNR) approach to firstly remove the potential mislabels from the benign training dataset, and then conducted the malware prediction experiment with the RF approach. For simplicity, we call the RF based malware prediction approach as RF\_MP, call the LR-LR based malware prediction approach as LR-LR\_MP, and call the benign mislabeled removed malware prediction using RF as ALNR+RF\_MP.

4.4.1 Testing with Benign B-2016 Dataset

We first did a malware prediction experiment with the benign B-2016 dataset. Since the malware prediction approach needs to train and test on disjoint datasets, we use M-2016 and part of B-2016 for training, and used the rest of B-2016 for testing.

Particularly, we divided the benign dataset B-2016 into 5 folds, then trained a classifier using malware dataset M-2016 and each fold of B-2016, and tested the classifier with the remaining folds of B-2016. In this way, each sample in the benign dataset B-2016 is predicted 4 times and we ensemble all of the predictions using a threshold of 2, i.e., we consider a target sample is detected as malicious if it is classified as malicious by at least 2 classifiers. For each malware prediction approach, there will be 5 different ML models built to get the all-fold predictions.

We conducted parameter selection for each malware prediction approach using another set of training data with a similar number of samples used for the formal experiment. For example, we randomly sampled 1/5 of the data from B-2016 as the benign training data, and used the sampled benign training data and all malware M-2016 data for parameter selection. We used grid search with 5-fold cross-validation to conduct parameter selection for all ML approaches. The number of trees for RF is 200 with $\sqrt{n}$ number of features; the $\tilde{p}$ (expected proportion of positives in the
unlabeled dataset) and $\lambda_U$ (label regularization parameter) parameters for LR-LR are 0.1 and 3, respectively; the $C$ parameter for standard SVM with linear kernel $^1$ is 0.5, which is used for the two classifiers in ALNR. The above optimal parameters are used in both the malware prediction approach and mislabel identification approaches.

Since all samples in B-2016 are benign according to the April 2016 VirusTotal scanning results, all of the predicted malicious samples can be considered as potential zero-day detection results, the reported zero-day detection results for each ML approach are shown in Table 4.2. We consider the December 2017 VirusTotal scanning reports for the B-2016 dataset as the upgraded ground truth label, and use the new ground truth labels to evaluate the zero-day detection results The overall zero-day malware detection results for different malware prediction approaches are shown in Table 4.2, where each entry represents the ensemble (e.g., all-fold) results for the corresponding approach. From the table, we can see that all malware prediction approaches can detect a substantial amount of potential and confirmed zero-day malware samples. And cleaning the training dataset by removing the potential mislabels (e.g., using ALNR) can help detect more confirmed zero-day malware.

Table 4.2: Experiment results for B-2016 dataset using malware prediction techniques

<table>
<thead>
<tr>
<th>Approach</th>
<th>Potential Detection threshold $n=10$</th>
<th>Detection threshold $n=5$</th>
<th>Detection threshold $n=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confirmed</td>
<td>Unconfirmed</td>
<td>Confirmed</td>
</tr>
<tr>
<td>RF_MP</td>
<td>1561</td>
<td>457</td>
<td>1104</td>
</tr>
<tr>
<td>LR-LR_MP</td>
<td>12093</td>
<td>490</td>
<td>11603</td>
</tr>
<tr>
<td>ALNR+RF_MP</td>
<td>2174</td>
<td>512</td>
<td>1662</td>
</tr>
</tbody>
</table>

Table 4.3: Experiment results for U dataset using malware prediction techniques

<table>
<thead>
<tr>
<th>Approach</th>
<th>Potential Detection threshold $n=10$</th>
<th>Detection threshold $n=5$</th>
<th>Detection threshold $n=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confirmed</td>
<td>Unconfirmed</td>
<td>Confirmed</td>
</tr>
<tr>
<td>RF_MP</td>
<td>16179</td>
<td>4290</td>
<td>11889</td>
</tr>
<tr>
<td>LR-LR_MP</td>
<td>31366</td>
<td>4806</td>
<td>26560</td>
</tr>
<tr>
<td>ALNR+RF_MP</td>
<td>20963</td>
<td>4981</td>
<td>15982</td>
</tr>
</tbody>
</table>

Comparing with previously reported Android malware detection results with regard to the detection of confirmed malware, the accuracy for zero-day malware detection is relatively low.

$^1$Other kernels are not used since they are not as efficient as linear kernel and their performance improvements are very small.
However, one must consider that we are training with knowledge only obtainable from April 2016, when none of the AV products was able to detect any of the later confirmed zero-day malware at that time. We thus view it as positive result, since it not only shows an ML based system can be used to achieve verifiable zero-day malware detection, but also reveals the challenging reality for the problem (e.g., an analyst may need to further review a substantial number of candidates to identify the real zero-day malware).

In summary, this experiment shows that the malware prediction based detection approaches indeed detect a substantial amount of zero-day samples which are confirmed by other AV products about one and a half years later. For example, if considering the samples that are detected by at least 2 AV products as “confirmed” malware, then about 50% of the potential zero-day malware detections are indeed malicious for RF,MP approach and ALNR+RF,MP approach. The LR-LR approach generates a higher number of confirmed zero-day samples, but also outputs dramatically larger number of unconfirmed detections, which results in lower detection accuracy.

4.4.2 Testing with Unknown U Dataset

We conducted malware prediction based zero-day detection experiment with unknown U dataset. The high quality malware dataset M-2016 and benign dataset B-2016 were used for training, and the samples in unknown dataset U were used for testing. For this experiment, we tentatively assign “benign” labels for the unknown dataset U and consider the tentatively assigned label as (noisy) version-1 labels.

In order to reduce the training size and create a training dataset with an approximate malware to benign class ratio of 1:10, we again split the benign dataset B-2016 into 5 folds, then trained a classifier using the malware dataset M-2016 and each fold of B-2016, and tested the classifier with all the samples in U. In summary, each sample in the unknown dataset U is predicted 5 times and we ensemble the predictions using a threshold of 2 to get the final detection results.

We conducted parameter selection for each malware prediction approach using another set of training data that has a similar number of samples used for the formal experiment, e.g., we randomly sampled 1/5 of benign data from B-2016 as benign training data, and used all malware M-2016 data. Grid search with 5-fold cross-validation was used to conduct parameter selection for
all ML approaches. The number of trees for RF was 450 with sqrt number of features; the $\tilde{p}$ and $\lambda_U$ parameters for LR-LR were 0.1 and 3, respectively; the $C$ parameter for standard SVM with a linear kernel was 8, which was used for the two classifiers in ALNR.

For all the potential zero-day samples, we checked their May 2017 AV scanning results to get the confirmed zero-day malicious samples. We tried to scan the detected app if it has not been analyzed before and its size is within the VirusTotal specified size limit; and we reused the existing scanning report if the app had already been scanned. The overall zero-day detection results for unknown dataset $U$ using malware predictions approaches are shown in Table 4.3.

We can see that a significant amount of the ML system detected potential zero-days in $U$ were already confirmed as malicious by existing AV products by May 2017. Since the training dataset (i.e., $B-2016$) has higher quality than the testing dataset (i.e., $U$), we get a larger number of potential zero-day samples and confirmed zero-day detections. For example, about 70% of the potential zero-day detections are confirmed as malicious for RF_MP and ALNR+RF_MP, when considering the samples that are detected by at least 2 AV products as “confirmed” malware. This experiment shows that the malware prediction approaches can be used to achieve verifiable zero-day detection even when trained with old datasets.

4.5 Mislabel Identification Experiment

We present the experiment to achieve verifiable zero-day malware detection using mislabel identification techniques.

Table 4.4: Experiment results for $B-2016$ dataset using mislabel identification techniques

<table>
<thead>
<tr>
<th>Approach</th>
<th>Potential Zero-days</th>
<th>Detection threshold $n=10$</th>
<th>Detection threshold $n=5$</th>
<th>Detection threshold $n=2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confirmed</td>
<td>Unconfirmed</td>
<td>Confirmed</td>
<td>Unconfirmed</td>
</tr>
<tr>
<td>RF_MI</td>
<td>535</td>
<td>270</td>
<td>265</td>
<td>360</td>
</tr>
<tr>
<td>LR-LR_MI</td>
<td>10617</td>
<td>444</td>
<td>10173</td>
<td>723</td>
</tr>
<tr>
<td>ALNR_MI</td>
<td>3706</td>
<td>341</td>
<td>3365</td>
<td>517</td>
</tr>
</tbody>
</table>

In this experiment, we use Random Forests (RF) and Label Regularized Logistic Regression (LR-LR) for the mislabel identification scenario, since malware prediction techniques can also be used for mislabel identification purpose when training and testing on the same dataset. Additionally, we use the Active Label Noise Removal (ALNR) for the mislabel identification experiment.
For simplicity, we call the RF based mislabel identification approach as RF_MI, call the LR-LR based mislabel identification as LR-LR_MI, and call the ALNR based mislabel identification as ALNR_MI. Since the main focus of the experiment is zero-day malware detection, the high quality malware dataset M-2016 is only used for training.

4.5.1 Testing with Benign B-2016 Dataset

We first conducted experiment with benign B-2016 dataset. Since the mislabel approaches train and test on the same dataset, we used M-2016 and B-2016 for training, and B-2016 for testing.

Table 4.5: Experiment results for U dataset using mislabel identification techniques

<table>
<thead>
<tr>
<th>Approach</th>
<th>Potential Zero-days</th>
<th>Detection threshold n=10</th>
<th>Detection threshold n=5</th>
<th>Detection threshold n=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Confirmed</td>
<td>Unconfirmed</td>
<td>Confirmed</td>
</tr>
<tr>
<td>RF_MI</td>
<td>61</td>
<td>17</td>
<td>44</td>
<td>20</td>
</tr>
<tr>
<td>LR-LR_MI</td>
<td>2751</td>
<td>690</td>
<td>2061</td>
<td>838</td>
</tr>
<tr>
<td>ALNR_MI</td>
<td>1606</td>
<td>568</td>
<td>1038</td>
<td>669</td>
</tr>
</tbody>
</table>

In order to reduce the training size and create a training dataset with an approximate malware to benign class ratio of 1:10, we divided the benign dataset B-2016 into 5 folds, and created ML model with the malware M-2016 and each fold of B-2016, and apply the ML model on current fold of B-2016 to get the mislabels from each fold of the benign dataset. The final mislabel identification results are the sum of all mislabeled samples from all folds. For this experiment, we used the same data split and optimal parameters as used in Section 4.4.1.

Similar to the malware prediction experiment, we directly report the identified mislabel samples in B-2016 as the potential zero-day malware detection results. We evaluated the zero-day detection results according to the December 2017 VirusTotal scanning reports, and the overall mislabel identification based zero-day detection results are shown in Table 4.4, where each entry represents the all-fold combined result for the corresponding approach.

For the three ML approaches, LR-LR outputs a similar number of reported zero-day detections and confirmed zero-day detections; RF generates a lower number of reported zero-day detections, but the ratio of the confirmed zero-day detections are increased from the malware prediction scenario; using ALNR for mislabel identification based malware detection generated a larger
number of potential zero-day samples, but the ratio of confirmed zero-day detections are smaller than the ALNR+RF approach.

Overall, Table 4.4 shows that mislabel identification approaches can indeed be used for malware detection, but they generate a slightly lower number of confirmed zero-day detections than malware prediction approaches. We believe this is because the benign training dataset (i.e., dataset B-2016) used by malware prediction approaches is much better than the benign training dataset (i.e., dataset U) used by the mislabel identification approaches.

4.5.2 Testing with Unknown U Dataset

We conducted mislabel identification based zero-day detection experiment with unknown dataset U. For this experiment, we tentatively assign “benign” labels to all the unknown samples, and consider the tentatively assigned label as (noisy) version-1 labels.

In order to reduce the training size and create training dataset with approximate class ratio of 1:10, we split unknown dataset U into 3 folds, then compared the malware dataset M-2016 against each fold of the unknown dataset, and identified the mislabels within each fold of U.

We also conducted parameter selection using another set of training data with similar number of samples used for the formal experiment. In particular, we randomly sampled 1/3 of data from U as “benign” training data, and used all malware M-2016 data for parameter selection. We used grid search with 5-fold cross-validation to conduct parameter selection for all ML approaches. The number of trees for RF was 500 with \( \sqrt{\text{number of features}} \); the \( \tilde{p} \) and \( \lambda_U \) parameters for LR-LR were 0.1 and 3; the \( C \) parameter for standard SVM with linear kernel was 32, which was used for the two classifiers in ALNR.

To verify the maliciousness of the potential zero-day malicious samples, we scanned the detected samples against VirusTotal in May 2017. The mislabel identification based zero-day detection results for unknown dataset U are also shown in Table 4.5. We can see that even though the labels for the benign training dataset used in this experiment are very noisy (e.g., tentatively assigned), the ML systems can still report a substantial amount of verifiable zero-day detections.

Comparing with malware prediction approaches, the three mislabel identification approaches both generate a dramatically smaller number of total malware detections and potential zero-day
## Table 4.6: Manually confirmed zero-day malware samples

<table>
<thead>
<tr>
<th>Family Behavior</th>
<th>SHA256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware Dropper I.</td>
<td>Redirect user to an app downloading website (<a href="http://www.hongtaoq.com/tuijian.html">http://www.hongtaoq.com/tuijian.html</a>), and download an malicious app which has been confirmed as malware by more than 10 AV vendors on VirusTotal.</td>
</tr>
<tr>
<td></td>
<td>5584c9aa9d975a4a80d4864bb29d443e832e13729f62e8784741dd31a558457</td>
</tr>
<tr>
<td></td>
<td>66c0e121385e7e0d63f5887f8a834b8f78d7e9c53b0861557f342686</td>
</tr>
<tr>
<td></td>
<td>42085af41f9326569c091931e1e80648e5ec09260eb276e21e6a456aa63b2</td>
</tr>
<tr>
<td></td>
<td>8e8829f4a1a65f64e4a32b88726e7a8999e68309f142b55a521567921b8c</td>
</tr>
<tr>
<td></td>
<td>427c3f805b352b3f32689d6fde671b8241e5b709e5115eb169a312761107b41</td>
</tr>
<tr>
<td></td>
<td>e07979f29d1124cc252a81f58e4a1f63b3a885b31f727d115580b265a5811a1</td>
</tr>
<tr>
<td></td>
<td>0c92c9260a745a8e6e46f0c37104c241b1b0023504b0c927d435851127</td>
</tr>
<tr>
<td></td>
<td>21097837f15786ed31c1d7bece442158ed28434c21e6c304994117267f6eca</td>
</tr>
<tr>
<td></td>
<td>392682a4271f948c100de836a36a2068a79bce184a5a524847ecba46925247</td>
</tr>
<tr>
<td></td>
<td>816ba4022e22d9a29da5d1975e6458dcb24de2f76dd1f7e7bd924e9b87c8</td>
</tr>
<tr>
<td>Malware Dropper II.</td>
<td>Keep a service running in the background to fetch apk url list from a C&amp;C server. Download and install those apks on user’s device and automatically launch them after installation.</td>
</tr>
<tr>
<td></td>
<td>090ae6df94c0077853fa577dd6b15a5a18e855e353d8ad1841f8332ce9ba020f</td>
</tr>
<tr>
<td>Information Stealer I.</td>
<td>Stealthily send personal info to remote server, receive commands from C&amp;C server to send text message, make phone call, etc.</td>
</tr>
<tr>
<td></td>
<td>1aaee3a6c4ee887f199943375a6c5ab26815b3c5606dfb4d11bf53444b5ced74</td>
</tr>
<tr>
<td></td>
<td>86ac7f54e87c6d26f0190255066a611a6e08f0a82b1a94b633506788527</td>
</tr>
<tr>
<td>Information Stealer II.</td>
<td>Stealthily upload contacts, phone/voice records, and device info to remote server.</td>
</tr>
<tr>
<td></td>
<td>835c112e59214e9ffbaea3ddb548d119ab358e368a00d1b77494a25e4ab9</td>
</tr>
<tr>
<td>Malicious Loan App.</td>
<td>Stealthily upload phone records, text and upload SMS messages to remote server.</td>
</tr>
<tr>
<td></td>
<td>9a4d4d9e1e5e2d2109286d57a1ac80847c290396a296d9898843877471f</td>
</tr>
<tr>
<td>Aggressive Advertisement.</td>
<td>Launch a service to keep showing ads, when “onDestroy” event received by this service, it will schedule to relaunch itself.</td>
</tr>
<tr>
<td></td>
<td>8f650df8460ca179807e5fa8e9461039bb83f4bcb721a1267a39e8437d71a</td>
</tr>
</tbody>
</table>

The potential zero-day detection results also indicates that for zero-day malware detection with a completely unknown dataset, training with a separate benign dataset using a malware prediction approach is preferred than directly training with part of the target dataset using a mislabel identification approach. However, since the resulting ML model will improve if trained with cleaner dataset, we conjecture the mislabel identification approach should be able to detect more zero-day samples if repeatedly applied.

### 4.6 Manual Verification of Zero-Day Malware

For the potential zero-day samples in unknown dataset U that were still not detected as malicious by any AV products by May 2017, we performed manual analysis (with the help of static and dynamic analysis tools) to obtain the ground truth of their maliciousness. Since it will be infeasible to manually go through all the potential zero-day malware apps, we randomly selected 30 unconfirmed malware samples from each ML detection approach, and obtained a total of 176 unique samples (after removing 4 duplicated samples) for analysis. From the prepared candidate samples, we found 8 apps which we believe are real world zero-day malware. Note that we excluded detections.
the border line apps that were only collecting non-critical information for various purposes. The identified zero-day malware samples can be categorized into 5 families and the detailed malicious characteristics for each family are shown in Table 4.6.

Note that we conducted the zero-day detection experiments (e.g., Section 4.4.2 and Section 4.5.2) with unknown dataset U around May 2017. We mainly used the 2016 dataset for training since a majority of our confirmed malicious samples were collected before April 2016 and their AV scanning results are more stable, while newer Android apps typically do not have large number of malicious AV labels yet. For completeness, we scanned the identified zero-day samples in VirusTotal in February 2018, and only one of them was confirmed as malicious. The manually confirmed malware samples also show that we can use an ML detection system to achieve real world zero-day malware detection.

4.7 Summary

In this chapter, we proposed to evaluate machine learning by checking zero-day malware detection, which means applying machine learning techniques to find unknown knowledge from existing knowledge. Besides the traditional malware prediction approaches, we also examined the mislabel identification approaches. Experiments show that both malware prediction and mislabel identification approaches can be used to achieve verifiable zero-day malware detection, even when trained with an old and noisy ground truth dataset. Using an ML based approach to analyze real world unknown Android apps, we manually confirmed 19 zero-day malware samples from 176 representative potential zero-day detection results, 18 of them were still not detected as malicious in VirusTotal by February 2018.
CHAPTER 5: CONCLUSIONS

Nowadays, malware remains one of the major security threats for both individuals and organizations. Similarity analysis techniques play an important role for effective malware triaging and malware family detection. In this dissertation, we worked on several areas in similarity based large scale malware analysis.

In Chapter 2, we proposed an effective binary similarity analysis tool called BingSim based on a novel CFG similarity comparison algorithm utilizing blended n-gram graphical features. BingSim converts binaries from the graphical features extracted from the corresponding CFGs into a fixed-size fuzzy hash output, which is as compact as cryptographic hash SHA-2 and can be easily compared through simple bitwise comparison. Our comprehensive evaluation demonstrates that the underlying CFG comparison algorithm CFGSimNG is more stable, faster, and generates more accurate results compared to five state-of-the-art CFG comparison algorithms, and BingSim effectively performs malware clustering and malware detection tasks with 2865 carefully labeled malware samples in an efficient manner.

In Chapter 3, we discuss the design and implementation of an Android malware clustering system through iterative mining of malicious payload and checked whether malware samples share the same version of malicious payload. In order to distinguish between the legitimate Java libraries and bogus libraries which contain malicious payloads, we designed a novel method to precisely remove legitimate library code from Android apps while preserving the malicious payloads even if they are injected under popular library names. We conducted extensive experiments to evaluate the consistency and robustness of our clustering solution. Our experimental results demonstrate that our clustering approach achieves precision of 0.90 and recall of 0.75 for the Android Genome malware dataset, and an average precision of 0.984 and recall of 0.959 regarding manually verified ground-truth.
Finally, in Chapter 4, we studied the fundamental issues faced by traditional ML based Android malware detection systems, examined the role of ML based for Android malware detection systems in practice, which leads to a revised evaluation strategy that evaluates ML based malware detection system by checking their zero-day detection capabilities. Through extensive experiments, we demonstrate that: (a) it is feasible to evaluate ML based Android malware detection systems with regard to their zero-day malware detection capabilities; (b) both malware prediction and mislabel identification approaches can be used to achieve verifiable zero-day malware detection, even when trained with an old and noisy ground truth dataset.

Even only considering similarity based malware analysis, numerous analytic systems have already been proposed. And there are more and more anti-virus products released in recent years. However, real world malware detection solutions are still far from perfect, and there are still several areas one can extend in the future.

- **Increasing the fidelity of malware detection results:** More and more security companies are developing their own anti-virus products, and end users in theory should benefit from diversified malware detection capabilities, in that the real malware will always be captured by certain detection solutions. However, the reality is that users often found the malware detection results are not immediately actionable for the following reasons: (1) the term “malicious” is extremely overloaded in the context of malware detection, *e.g.*, certain samples may cause severe financial damage while others maybe just cause mildly annoyance; (2) a significant number of anti-virus products are built with machine learning models or heavily rely on heuristics, *e.g.*, directly outputting results like “suspicious (confidence 75%)” is trying to shift the maliciousness decision burden to end-users; (3) when considering multiple anti-virus products at the same time, we may get inconsistent malware detection results and there is no perfect threshold for deciding how many anti-virus products flagged malware as real malware. From an user perspective, a useful and successful anti-virus product should be able to answer with high certainty: whether the target sample is a real malware? and what family it belongs to? This seems to be the high level objective for anti-virus products as well, but
it is non-trivial to achieve in practice. Effective similarity analysis techniques maybe helpful, but only relying on such techniques won’t be enough.

• Reducing the detection false positives and false negatives No anti-virus products can confidently claim they are free of false positives and false negatives, especially for ML based malware detection solutions. A false negative output means it failed to identify the malicious sample. Since whether an anti-virus product can detect certain malware is a direct perception of its detection capability, a product with accumulated false negatives will be viewed as weak performance or unreliable. A false positive output means incorrectly detection benign sample as malicious. Because the number of benign samples are significantly larger than the number of malicious samples in real world, anti-virus products will become unusable if the absolute number of incorrect results are high (even though the false positive ratio may still be relatively low). Instead of trying to make a two-class (e.g., benign or malicious) decision, it maybe helpful to firstly separate in three categories: high confidence malicious, high confidence benign, uncertain; then drill down on the uncertain cases by conducting more heavy loaded analytics. Due to the multi-layer nature, such a system will be more successful if designed in offline mode.
REFERENCES


APPENDIX A: COPYRIGHT PERMISSIONS

This Agreement between Yuping Li ("You") and Springer Nature ("Springer Nature")
consists of your license details and the terms and conditions provided by Springer Nature
and Copyright Clearance Center.

License Number 4363110303472
License date Jun 06, 2018
Licensed Content Publisher Springer Nature
Licensed Content Publication Springer eBook
Licensed Content Title Android Malware Clustering Through Malicious Payload Mining
Licensed Content Author Yuping Li, Jiyong Jang, Xin Hu et al
Licensed Content Date Jan 1, 2017
Type of Use Thesis/Dissertation
Requestor type academic/university or research institute
Format print and electronic
Portion full article/chapter
Will you be translating? no
Circulation/distribution <501
Author of this Springer Nature content yes
Title Android Malware Clustering Through Malicious Payload Mining
Instructor name Yuping Li
Institution name University of South Florida
Expected presentation date Jun 2018
Requestor Location TAMPA, FL 33617

Billing Type Invoice
Billing Address TAMPA, FL 33617
United States
Attn: Yuping Li

Total 0.00 USD

Terms and Conditions

Springer Nature Terms and Conditions for RightsLink Permissions

https://s100.copyright.com/AppDispatchServlet
Springer Customer Service Centre GmbH (the Licensor) hereby grants you a non-exclusive, world-wide licence to reproduce the material and for the purpose and requirements specified in the attached copy of your order form, and for no other use, subject to the conditions below:

1. The Licensor warrants that it has, to the best of its knowledge, the rights to license reuse of this material. However, you should ensure that the material you are requesting is original to the Licensor and does not carry the copyright of another entity (as credited in the published version).

   If the credit line on any part of the material you have requested indicates that it was reprinted or adapted with permission from another source, then you should also seek permission from that source to reuse the material.

2. Where print only permission has been granted for a fee, separate permission must be obtained for any additional electronic re-use.

3. Permission granted free of charge for material in print is also usually granted for any electronic version of that work, provided that the material is incidental to your work as a whole and that the electronic version is essentially equivalent to, or substitutes for, the print version.

4. A licence for ‘post on a website’ is valid for 12 months from the licence date. This licence does not cover use of full text articles on websites.

5. Where ‘reuse in a dissertation/thesis’ has been selected the following terms apply: Print rights for up to 100 copies, electronic rights for use only on a personal website or institutional repository as defined by the Sherpa guideline (www.sherpa.ac.uk/romeo/).

6. Permission granted for books and journals is granted for the lifetime of the first edition and does not apply to second and subsequent editions (except where the first edition permission was granted free of charge or for signatories to the STM Permissions Guidelines http://www.stm-assoc.org/copyright-legal-affairs/permissions/permissions-guidelines/), and does not apply for editions in other languages unless additional translation rights have been granted separately in the licence.

7. Rights for additional components such as custom editions and derivatives require additional permission and may be subject to an additional fee. Please apply to Journalpermissions@springernature.com/bookpermissions@springernature.com for these rights.

8. The Licensor’s permission must be acknowledged next to the licensed material in print. In electronic form, this acknowledgement must be visible at the same time as the figures/tables/illustrations or abstract, and must be hyperlinked to the journal/book’s homepage. Our required acknowledgement format is in the Appendix below.

9. Use of the material for incidental promotional use, minor editing privileges (this does not include cropping, adapting, omitting material or any other changes that affect the meaning, intention or moral rights of the author) and copies for the disabled are permitted under this licence.

10. Minor adaptations of single figures (changes of format, colour and style) do not require the Licensor’s approval. However, the adaptation should be credited as shown in Appendix below.

**Appendix — Acknowledgements:**

For Journal Content:
Reprinted by permission from [the Licensor]: [Journal Publisher (e.g. Nature/Springer/Palgrave)] [JOURNAL NAME] [REFERENCE CITATION (Article name, Author(s) Name), [COPYRIGHT] (year of publication)