Large-Scale Malware Analysis, Detection, and Signature Generation

by

Xin Hu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Atul Prakash
Assistant Professor J. Alex Halderman
Assistant Professor Qiaozhu Mei

To my parents and many other important people in my life

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Professor Kang G. Shin, for his tremendous support, guidance and encouragement during the course of my doctoral study. His insights into research, values of life and commitment to excellence have inspired me significantly in my pursuit of research. I feel truly grateful to be his student and work with him for all these years. I would also like to thank Professors Atual Prakash, J. Alex Halderman, and Qiaozhu Mei for serving on my dissertation committee. Without their insightful comments and feedback, this dissertation would not be possible.

I am also grateful to all talented and friendly colleagues for their friendship and accompany during my graduate student life. Special thanks go to all the past and current members at Real-Time Computing Laboratory. In particular, I would like to thank: Zhigang Chen for being such a good friend in my life and his willingness to help with just about anything; Matthew Knysz for being an amazing collaborator and tremendous help in improving my writing; Yuanyuan Zeng for collaboration and pleasant conversations from time to time; Caoxie Zhang for the inspiring discussions on many research areas and interesting conversations in days and nights when we were working in the same office; Xiaoen Ju for his kindness and diligence in our collaboration; Xinyu Zhang, Alex Min, Katharine Chang, Karen Hou, Hyoil Kim, Ashiwini Kumar, Min-gyu Cho, Jisoo Yang, Pradeep Padala, Eugene Chai and others for their invaluable comments on my research. I would also like to thank Bin Liu, Joseph Xu, Simon Chen, Yi Su, Bengheng Ng, Ying Zhang, Ran Duan, Ye Du,

Fangjian Jin, and Zhe Chen for their wonderful friendship that makes my life in Ann Arbor exciting and memorable.

I was also very fortunate to work as an intern at Symantec Research Labs. Special thanks go to my mentors Professor Tzi-cker Chiueh and Kent Griffin for their detailed guidance and advice. My appreciation also goes to Dr. Sandeep Bhatkar, Scott Schneider, Darren Shou, and Fanglu Guo for support and collaboration. I also would like to thank Symantec Research Labs for their generosity in providing a large amount of malware samples for my dissertation studies and a graduate fellowship.

Finally, I want to express sincerest and deepest gratitude to my parents as well as special friends Yang Lin, Kun Qian and Mingming Hu for their unconditional love, support and encouragement that have enlightened and enriched my life. I cannot describe how thankful and fortunate I am to have them on my side through all the happiness, sadness and joys during these years.

The work described in this thesis was supported in part by the US Air Force Office of Scientific Research under Grant No. FA9550-10-1-0393 and the US Office of Naval Research under Grant No. N000140911042.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Large Scale Malware Analysis, Detection and Signature Generation

by

Xin Hu

Chair: Kang G. Shin

As the primary vehicle for most organized cybercrimes, malicious software (or malware) has become one of the most serious threats to computer systems and the Internet. With the recent advent of automated malware development toolkits, such as Zeus, it has become relatively easy, even for marginally skilled adversaries, to create and mutate malicious codes which can bypass Anti-Virus (AV) detection. This has led to a surge in the number of new malware threats and has created several major challenges for the AV industry. AV companies typically receive tens of thousands of suspicious samples every day, which have to be analyzed by human analysts in order to 1) determine the maliciousness of incoming samples; 2) identify their labels (e.g., family name); and 3) create AV signatures. However, the overwhelming number of new malware easily overtax the available human resources at AV companies, making them less responsive to new emerging threats and eventually leading to poor detection rates. To address the aforementioned issues, this dissertation proposes several new and scalable systems to facilitate malware analysis and detection, with the focus on a central theme: "automation and scalability".

This dissertation makes four primary contributions. First, it builds a large-scale

malware database management system called SMIT that addresses the challenges in the first step of processing each incoming suspicious sample, i.e., determining if it is indeed malicious. The system is based on the insight that most new malicious samples are simple syntactic variations of existing malware, and hence, a way to ascertain the maliciousness of an unknown sample is to check if it is sufficiently similar to any currently known malware. SMIT is designed to efficiently make such decisions based on the malware's function call graph—a high-level structural representation that is less susceptible to the low-level obfuscation employed by malware writers to evade detection. Evaluation of real-world malware samples demonstrates SMIT's effective pruning power and scalability to support hundreds of thousands of malware samples. Second, due to limited human resources, a large percentage of samples received by AV companies often remain unlabeled in the database for an extended period of time. To overcome this problem, the dissertation develops an automatic malware clustering system called MutantX. By quickly grouping similar samples into clusters, MutantX allows malware analysts to focus on representative samples from each cluster and automatically generate labels for unknown samples based on their association with existing groups. Third, this thesis introduces a malware signature-generation system, called Hancock, that automatically creates high-quality string signatures with extremely low false-positive rates. Finally, observing that two widely used malware analysis approaches—i.e., static and dynamic analyses—have their respective pros and cons, this dissertation proposes a novel system, called DUET, that optimally integrates malware clusterings based on both static features and dynamic behaviors. The goal of DUET is to allow the static and dynamic analyses to complement each other and mitigate their respective shortcomings without losing their merits.

CHAPTER I

Introduction

1.1 Background

As computer systems and the Internet become increasingly ubiquitous, the security threat landscape has also undergone a profound transformation from unstructured and sporadic attacks, where the primary intent is a quest for fame, to more organized multi-vector attacks on a global scale, where the goal is financial profits. The lack of sophisticated protection on average users' computers and the high value of enterprise targets have attracted skilled and motivated cyber-criminals to launch a wide range of security attacks. These attacks compromise computers, penetrate networks, steal confidential information, send out lots of spam emails, bring down servers and cripple critical infrastructures, leading to severe damage and significant financial loss. According to a recent CSI (Computer Security Institute) survey [49], the average loss from security attacks was about $345,000 per incident.

The main vehicle for most organized cyber crimes is various types of malware. Malware, or malicious software, generally refers to various forms of hostile, intrusive and annoying software designed to infiltrate a computer system and subvert the system for unintentional uses. Typical malware types include viruses, worms, spyware, trojan horses, rootkits, and bots. Spreading a destructive payload, they infect and take control of vulnerable computer systems, using them to facilitate other criminal

activities and gain illegal profit [97]. For example, bots typically spread through exploiting software vulnerabilities or employing social engineering techniques to allure unsuspecting users to execute malware binaries. Once a system has been infected, the malware can install spyware and backdoors, transforming these individual victimized systems into a vast network, called a *botnet*, controlled by the attackers. Botnets are commonly used in launching DDoS (Distributed Denial of service) attacks, sending spam emails and hosting phishing fraud.

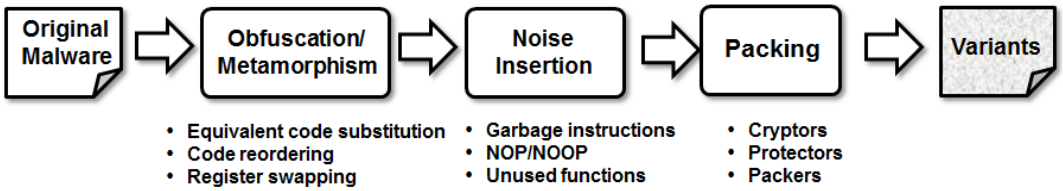


Figure 1.1: Production of malware variations

Driven by considerable economic incentives, both the diversity and the sophistication of malware have increased significantly. Malware has evolved from rudimentary viruses that delete system files to more versatile, highly engineered pieces of software that are able to carry out advanced, large-scale attacks. For example, Stuxnet, discovered in early 2010, is the first publicly known malware program targeting critical industrial infrastructure. It has the capability to infect high-value industrial control systems and inject code into the systems' PLC (Programmable Logic Controller) unit, allowing Stuxnet to potentially control the system, altering its operations [30]. However, despite this dramatic increase in malware complexity, the knowledge required by attackers to create malware threats has actually decreased substantially in recent years. This is due mainly to the growing popularity of easy-to-use, automatic malware creation toolkits, such as Zeus [103] or SpyEye [24]. Such toolkits allow even marginally skilled attackers to create and customize their own malware binaries, significantly lowering the novice attackers' barriers to enter the world of cyber crime

and resulting in a massive proliferation of new malware.

In addition, most malware programs are continuously mutated to evade anti-virus (AV) detectors. Instead of the time-consuming and expensive process of creating a malware program from scratch, malware authors often pursue a more cost-effective solution; reusing existing malware (either binaries or source codes) by slightly altering them to evade AV detectors. Because of this success, such malware variants have evolved into a streamlined process [79] where malware authors employ a broad spectrum of tools and technologies to automatically create variants capable of eluding detectors. Typical techniques (Figure 1.1) include equivalent code substitution, instruction re-ordering, noise insertion and runtime packing (i.e., encrypting or compressing the original binaries into random-looking data and decrypting the content when the malware is executed). The ability to automatically and rapidly create variants allows malware authors to replace outdated malware as soon as they become less effective, granting them an advantageous attack window before new detection signatures can be created and deployed. The ease of this malware-mutation process has led to an explosive increase in the number of new malware samples seen in the field, as shown in Figure 1.2. From the figure, we can observe that the number of malware has nearly doubled annually year-to-year basis, and the total number of new malware created in 2009 has reached 2.9 million, which is equivalent to over 8000 new variants appearing daily. In fact, the total number of malware programs produced in 2009 alone is more than the sum of all malware created over the previous 20 years. Unfortunately, this trend is likely to continue, and malware will remain the greatest security threat faced by computer users.

1.2 Motivation

The explosive increase in malware variants has created a key challenge for the AV industry: how to efficiently process this huge influx of malware samples and promptly

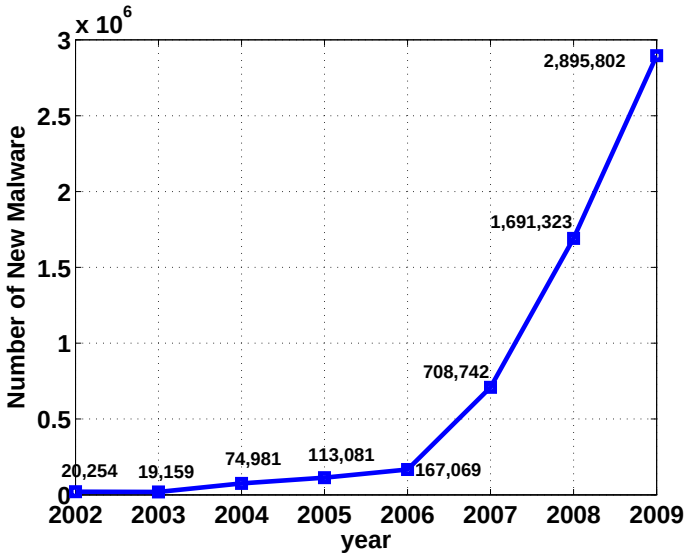


Figure 1.2: Exponential increase in the number of new malware samples (Source: Symantec [97])

deploy mitigation techniques to protect end-users. An AV company typically receives thousands of suspicious samples every day, collected from tools such as honeypots and global monitoring sensors [97] or submitted by their partners (e.g., other AV companies that share malware samples), clients and third-party collection channels [3, 107]. These suspicious samples are typically processed with the following steps.

S1. Malware analysts have to determine if the incoming suspicious samples are indeed malicious, separating malicious programs from benign ones.

S2. For malicious samples, analysts have to establish which malware family each sample belongs to, and then create family labels for these samples.

S3. New virus signatures have to be generated and distributed to end-users for their protection.

All the above processes require some level of human intelligence and are mostly done through manual analysis, which, unfortunately, is expensive, time-consuming and error-prone. The overwhelming number of new malware programs has severely

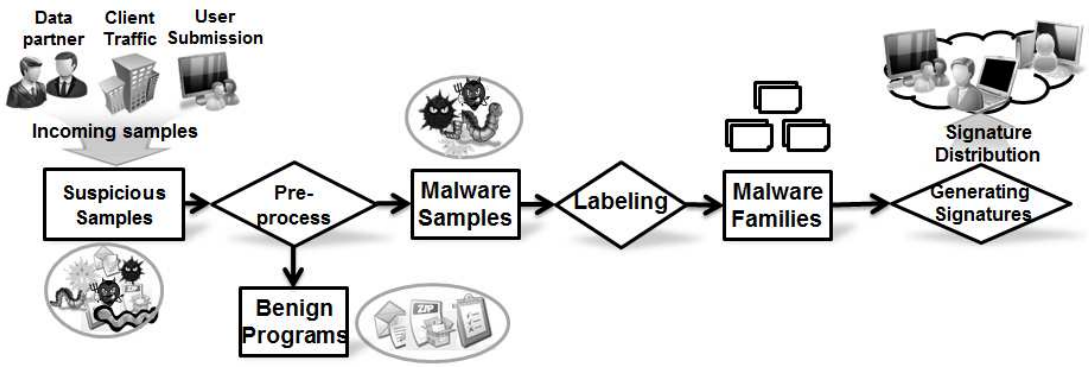


Figure 1.3: Typical malware processing workflow

strained the scarce human resources of AV companies, making them less responsive to new threats and even allowing some malware to slip through and remain undetected for a significant period of time. For instance, there is a typical time window of 54 days between a malware's release and its detection by AV software, and 15% of samples remain undetected after 180 days [27]. As a result, manual analysis has become the major bottleneck in the malware processing workflow, calling for automatic techniques to analyze incoming samples and produce high-quality signatures. Such techniques can allow AV vendors to keep up with rapid malware generation and deployment, reducing their response time to new security threats.

1.3 Malware Analysis

In general, there are two types of approaches commonly used to analyze malware programs: *dynamic behavior* and *static feature* based analyses. Most dynamic analysis systems operate by running malware samples in virtual or sandboxed environments, monitoring their execution and extracting their run-time behavior in terms of API or system call traces for analysis and detection [10, 13, 88]. On the other hand, static analysis systems do not require running the malware programs; they extract representative features directly from malware binaries [81, 113] or from disassembled

instructions [53, 56]. Some times, high-level structural features, such as control flow graphs or function call graphs, [29, 32] can also be extracted from the disassembled instructions and used as a basis for malware analysis.

The major benefit of dynamic analysis is that behavioral features are insensitive to low-level mutation techniques, such as run-time packers or binary obfuscation, because changes to a malware's binary rarely affect API or system calls it invoked. In addition, behavioral-features-based detection identifies *actions* performed by malware rather than syntactic signatures, thus having the potential to capture multiple malware family variants with a single behavioral specification. Albeit very useful in practice, approaches based on dynamic behavioral features also suffer from several limitations. First, they may have only limited coverage of an application's behavior, failing to reveal the entire capabilities of a given malware program. This is because, when monitoring an executed malware program, dynamic analysis can only capture API or system call traces corresponding to the code path taken during that particular execution. However, different code paths may be taken during different executions, depending on the program's internal logics and/or external environments. More commonly, many malware include triggers in their programs, exhibiting an interesting behavior only when certain conditions are met. Typical examples include bot programs that wait for commands from their botmasters and malware programs designed to launch attacks at, or before, a certain date and time. Since their specific conditions are often not met, when executed and monitored in a general environment, these trigger-based malware generate few repeatable run-time traces. Second, dynamic-analysis is inherently resource-intensive, limiting its coverage. In order to process the sheer number of malware samples collected daily, a dynamic-analysis system, with limited computational resources, can only execute and monitor each sample for a short duration, e.g., a couple of minutes. Unfortunately, this is often too short a period for most malware programs to reveal all their behavior.

In contrast, the main advantage of static analysis is its potential to cover all possible code paths of a program, including parts of the program that normally do not execute, thereby yielding a more accurate characterization of the program's entire functionalities. Moreover, being less resource-intensive and time-consuming than their dynamic counterpart, the static approaches provide the level of scalability necessary for handling the rapid generation of new malware. Unfortunately, static-feature-based approaches are not without limitations of their own; it is well-known that they suffer from run-time packing and many anti-reversing and anti-disassembly techniques [117], such as encryption, compression, garbage code insertion, code permutation, etc.

Because of their respective pros and cons, neither dynamic behavior nor static feature based approaches provide a complete solution to the malware-analysis problem. In this dissertation, in order to scale and support the enormous number of malware samples, we first develop static approaches, with particular emphasis on novelty, addressing inherent limitations of static analysis. We also investigate ways to systematically integrate dynamic and static approaches, exploiting their respective strengths and mitigating their weaknesses. The underlying purpose of this combined method is to improve the coverage, accuracy and efficacy of malware analysis.

1.4 Research Goals

Motivated by the trend of large-scale threats, this dissertation develops solutions to automate the key stages in the malware processing workflow that have been traditionally done by human experts, facilitating large-scale analysis and detection of security threats. These solutions are developed with the following objectives.

- **Expediting identification of malicious samples:** AV companies receive thousands of suspicious program samples every day. The first step in processing any sample is to determine if it is indeed malicious. Currently, a common

approach is to classify a sample as malware if a sufficient number of existing commercial AV scanners consider it malicious. Although this approach is useful, it does not completely solve the problem; at any point in time, a significant percentage of new samples are unknown to existing AV scanners. Typically, these unknown samples are manually analyzed by security analysts, a time-consuming process that has become a major bottleneck in the malware processing workflow. Therefore, it is necessary to automate the handling of these rapidly increasing malware programs. One of our goals toward meeting this need is to enable efficient and scalable identification of malicious programs by using their structural similarity to existing malware programs.

- **Creating labels for unknown malware samples:** Given the excessive number of malware programs and the limited computing and human resources available, a large percentage of new malware samples often remain unlabeled in the database for an extended period of time. This delays the creation and distribution of signatures, resulting in poor detection rates. This dissertation aims to address the malware-labeling problem through malware clustering. Quickly and automatically clustering malware samples allows analysts to focus on more important and distinct samples instead of wasting their precious time on similar variants. For instance, one can group similar samples into a cluster and label them with high accuracy by analyzing only a few representative samples from the cluster. Moreover, the label of a new sample can be automatically derived if it is determined to belong to a known cluster. This study also intends to develop a generic unpacking technique for statically analyzing packed programs.

- **Enabling automatic generation of string signatures:** Scanning files for signatures has been a proven approach used in many commercial anti-malware products because of its extremely low false-positive rate. However, the size of

the signature database has grown significantly with the exponential increase in the number of new malware samples in recent years. One way to address this signature-explosion problem is to use string signatures, each of which corresponds to a contiguous byte sequence meant to match variants of a malware family rather than a specific malware program, thus resulting in a smaller signature set. However, most of the string signatures used today are created manually because it is difficult to automatically determine which byte sequence in a malware binary is less likely to generate false-positives. To address this problem, this study develops a practical and automated framework that tackles the problem of generating high-quality string signatures on a large scale.

- **Improving integration between static and dynamic analyses**: Static and dynamic approaches are both valuable tools for malware analysis and are widely used in practice. However, they both have inherent limitations as described in Section 1.3. As a result, malware samples that can be effectively analyzed by these two approaches are usually very different. This makes it very difficult to select a single best algorithm for malware analysis. To exploit their strengths and mitigate their weaknesses, this study aims at developing a unified framework that aggregates the results generated by static and dynamic approaches, improving the performance of analysis of a wide range of malware samples.

1.5 Contributions

To meet the aforementioned research goals, we design and implement prototype systems that characterize the inherent features of malware programs and exploit them for quick and accurate analysis. The key features of the proposed solutions are *automation* and *scalability*, which are imperative to cope with rampant malware and other security threats. The applicability and efficiency of these systems are demon-

strated through experimentation on more than 100,000 real-world malware samples. In the rest of this section, we summarize the major contributions of this dissertation.

1.5.1 Large-Scale Malware Indexing Using Function Call Graphs

In this dissertation, we build a system called `SMIT` (Scalable Malware Indexing Tree) that attempts to speed up the process of identifying the maliciousness of a suspicious sample. The system is based on the insight that since most new malware samples are simple syntactic variations of existing malware, one way to ascertain whether an incoming sample is malicious is to check if it is sufficiently similar to any currently known malware. `SMIT` can efficiently make such decisions based on the malware's function-call graph, a high-level structural representation known to be less susceptible to the low-level obfuscations employed by malware writers to evade detection. To improve the speed of graph comparison, we develop a polynomial-time graph-similarity algorithm by exploiting common sub-structures in malware call graphs. The algorithm closely approximates the inter-graph edit distance while reducing the computational complexity to $O(n^3)$. Furthermore, `SMIT` employs a multi-resolution indexing scheme to solve the scalability issue related to the graph database search. The scheme uses a computationally economical feature vector for early pruning and resorts to a more accurate—but computationally expensive—graph similarity function only when it needs to pinpoint the most similar neighbors. The unique combination of these techniques affords `SMIT` significant pruning power and allows it to easily scale to support hundreds of thousands of malware samples.

1.5.2 Malware Clustering based on Static Features

The current lack of rapid, automatic labeling of the massive number of malware samples seen daily delays the distribution of malware signatures, thus lowering the rate of malware detection and failing to detect rampant malware in the wild. Conse-

quently, there is a strong need to automatically cluster malware programs, enabling analysts to make informed decisions and prioritize which samples require the most attention. This dissertation proposes a framework, called `MutantX`, that is designed to perform efficient clustering of a large number of samples into families based on static features, i.e., code instruction sequences. This is motivated by the observation that if malware programs share common traits in their code instructions, they are likely derived from the same code base, and thus, from the same families. `MutantX` features a unique combination of several novel techniques to address the practical challenges of malware clustering. Specifically, it exploits the instruction format of x86 architecture and represents a binary program as a sequence of opcodes, facilitating the extraction of N-gram features. It also utilizes a hashing trick, recently developed in the machine learning community to reduce the dimensionality of the extracted feature vectors, significantly reducing the memory cost and computational complexity of clustering. Our comprehensive evaluation on a `MutantX` prototype using a database of more than 100,000 malware samples has shown that it can correctly cluster more than 80% of input samples within 2 hours, achieving a good balance between accuracy and scalability.

1.5.3 Automatically Creating String Signatures for AV Detection

The dissertation also develops the first automatic string-signature generation system, called `Hancock`, that takes a set of malware programs as input and automatically creates string signatures with extremely low false-positive rates and maximum coverage. The main challenge faced by `Hancock` is the difficulty in ensuring that its generated string signatures have sufficiently a low false-positive rate without accessing all the goodware in the wild. Based on the results of studying many false-positive signatures, the dissertation proposes several novel techniques that effectively overcome the false-positive problem of machine-generated string signatures: a scalable good-

ware modeling technique that can accurately estimate the occurrence probability of arbitrary byte sequences in goodware programs, a content-aware string signature candidate selection algorithm that checks if a candidate is part of a library function or sufficiently unique, and a set of diversity-based string signature filtering techniques that estimate a candidate's false-positive likelihood based on the similarity among the malware files it covers. Incorporating these techniques together into the `Hancock` prototype, we conduct experiments on numerous real-world malware programs and demonstrate that the string signatures automatically generated by `Hancock` can indeed meet the false-positive rate requirement of 0.1%. Finally, we improve `Hancock` by adding the capability to generate multi-component string signatures, where multiple disjoint byte sequences make up a single string signature. These signatures are more effective than traditional single-component string signatures in terms of both coverage and false-positive rate.

1.5.4 Integrating Static and Dynamic Analyses

To investigate the feasibility and the potential of integrating static and dynamic malware analyses, we develop `DUET` for malware clustering based on the concept of *clustering ensembles*. The basic idea is to combine the clustering results from both types of approaches so that they complement each other, reducing the respective limitations of dynamic and static analysis alone. For instance, static analysis cannot process malware programs packed with sophisticated packers, which are mostly irrelevant to dynamic analysis. On the other hand, a dynamic approach often fails to properly handle trigger-based malware programs or those that detect virtual environments (e.g., VMware and Qemu); in these instances,static analysis could easily extract features from all the code paths without being confused by the evasion techniques. As a result, the proposed techniques provide a way to represent the consensus across multiple clustering algorithms and account for their effectiveness on different

types of data. The outcome of this hybrid approach is a set of malware clusters of higher quality and with better coverage than those created by a single approach alone. To achieve this goal, we first build an automatic malware behavior monitoring framework, taking advantage of VMware VIX API [109] and system-call hooking tools such as Strace [16]. The framework enables automatic execution of malware binaries inside virtual machines, collecting all system calls with detailed arguments and enabling a comprehensive view of malware behavior. Then, the goal of `DUET` is to systematically integrate two sets of clusterings (i.e., from static and dynamic analyses) into a coherent set. The main challenge is that the number and shape of the clusters provided by the individual solutions may vary with the type of clustering methods and their particular view of the data. Even worse, the resulting clusterings may be contradictory to each other. To address these challenges, we exploit *clustering ensembles* [95] and *clustering quality assessment* [1] to reconcile the differences between clustering algorithms. Our comprehensive experimental results demonstrate that `DUET` is able to improve coverage by 20–40%, while keeping the precision near the optimal achievable by any individual clustering algorithm alone.

1.6 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter II describes our approach for efficiently indexing a prodigious malware graph database, together with the description and evaluation of the prototype `SMIT`. In Chapter III, we present and evaluate a static-feature-based analysis system, `MutantX`, for efficiently clustering a large number of malware programs into families. Chapter IV provides the detailed design and implementation of the automatic string-signature-generation system, `Hancock`, and demonstrates its applicability on real-world malware samples. Chapter V presents `DUET`, which efficiently integrates static and dynamic clustering approaches. Finally, Section VI concludes this dissertation.

CHAPTER II

SMIT: Large-Scale Malware Indexing Using Function-Call Graphs

2.1 Introduction

With the advent of automated malware development toolkits, creating new variants of existing malware programs to evade the detection of anti-virus (AV) software has become relatively easy even for un-skilled aggressors. This has led to a huge surge in the number of new malware threats in recent years. According to Symantec's latest Internet Threat Report [97], the company received 499,811 new malware samples in the second half of 2007 alone. The first step to process any received malware sample is to determine if the sample is indeed malicious. Currently, this step is largely done manually and thus is a major bottleneck of the malware processing workflow. Because most new malware samples are variants of previously-known samples through mutation of their source or binary code, one way to ascertain the maliciousness of a sample is to check if the sample is sufficiently similar to any previously-seen malware program. We describe the design, implementation and evaluation of a graph-based malware database management system, called `SMIT` (Symantec Malware Indexing Tree) that is developed specifically to perform such checks efficiently.

Most existing malware-detection methods treat malware programs as sequences of

bytes, and ignore their high-level internal structures, such as basic blocks and function calls. These methods are generally ineffective against recent malware threats for the following reasons. First, since most modern malware programs are written in high-level programming languages and compiled into binaries, a minor modification in source codes can lead to a significant change in binary codes. Second, the availability of automated obfuscation tools that implement techniques such as instruction reordering, equivalent instruction sequence substitution, and branch inversion, allows malware writers to easily generate new malware versions that are syntactically different from, but semantically equivalent to, the original version.

One way to overcome the difficulties of recognizing syntactically different and semantically identical variants of a malware program is to base the recognition algorithm on a high-level structure that is less susceptible to minor or local modifications. One example of such high-level structure is a program's function-call graph, which abstracts away byte- or instruction-level details and is thus more resilient to byte- or instruction-level obfuscations commonly employed by malware writers or malware development tools. Moreover, because a program's functionality is mostly determined by the library or system calls it invokes, its function-call graph provides a reasonable approximation to the program's run-time behavior. Therefore, the function-call graphs of the malware variants that are derived from the same source or binary code are often similar to one another. By representing each malware program in terms of its function-call graph, we translate the problem of finding a malware sample's closest kin in a malware database into one that searches for a graph's nearest neighbor in a graph database.

Our work is unique and different from the previous work on graph database query processing for the following three reasons. First, most previous graph database research focused on exact graph or subgraph matching, which requires a solution to the graph or subgraph isomorphism problems (both are well-known NP problems).

However, since malware variants are rarely subgraphs of one another, exact graph or subgraph matching is too restricted to be useful for identifying malware variants. Instead, SMIT supports graph-similarity search, which, given a query graph, pinpoints graphs in a database that are most similar to the query graph. Second, because the cost of computing a graph-similarity score, for example, the graph-edit distance, is exponential in the number of nodes/edges, most existing graph-similarity query methods assume that the number of nodes in the graphs is on the order of 10s. They are not directly applicable to SMIT because the number of nodes in a malware's function-call graph ranges from 100s to 1000s. For example, a variant of the Agobot family has 2,759 nodes and 5,851 edges in its function-call graph. Third, many existing graph-similarity query processing methods cannot scale to a large graph database; their applicable size are mostly on the order of 1000s. Considering the enormous number of malware samples that the AV industry receives every year, the main goal of SMIT is to support efficient similarity queries for databases of the size that is at least 100,000 and up to a million.

SMIT features a unique combination of techniques to address the scalability challenge associated with graph-similarity search. First, SMIT incorporates a polynomial-time graph-similarity computation algorithm whose result closely approximates the inter-graph edit distance. This algorithm exploits the structural and instruction-level information associated with the malware programs underlying the input graphs. Second, SMIT applies an optimistic vantage point tree [21] to index a graph database to speed up nearest-neighbor graph-similarity search. Third, SMIT employs multi-resolution indexing that uses a computationally economical feature vector for early pruning and resorts to a more accurate but computationally more expensive graph similarity function only when it needs to pinpoint the most similar neighbors. We have successfully built a SMIT prototype and tested its performance using a test database containing more than 100,000 distinct malware programs. Our evaluation results

demonstrate that SMIT exhibits effective pruning power and scales to large graph databases in that the query service time grows slowly with the number of graphs in the database.

The remainder of this chapter is organized as follows. Section 3.2 reviews previous related work on graph-database search and indexing. Section 2.3 and Section 2.4 present SMIT's graph-similarity algorithms, which is based on the Hungarian method[62] and exploits properties of the underlying malware programs. Section 2.5 describes the multi-resolution indexing scheme used in SMIT. Evaluation results for the current SMIT prototype are presented in Section 2.6. Section 2.7 discusses SMIT's limitations and Section 2.8 concludes this chapter.

2.2 Related Work

Most existing work detects or classifies malware based on either byte-level signature [23] or malware run-time behavior [11, 14]. For example, Kolter and Maloof used n-gram of byte codes as features to train the classifier [57]. Rieck *et al.* [87] monitored the malware behavior (e.g., changes to file system and registry) in a sandbox and used supervised learning to predict malware families. Lee and Mody [101] collected sequences of system-call events and applied clustering algorithms to group malware families. Bailey *et al.* [11] defined malware behavior as non-transient state changes on the system and applied hierarchical clustering algorithms for malware grouping. More recently, Bayer *et al.* [14] applied Locality Sensitive Hashing on the behavior profiles to achieve efficient and scalable malware clustering. Both signature- and behavior-based approaches have their own limitations. The former is vulnerable to obfuscation and ineffective in identifying new malware samples. The latter, on the other hand, incurs expensive runtime overhead and tends to generate many false positives. SMIT differs from both in that it builds a large malware database based on their function-call graphs and supports efficient indexing techniques that allow

malware analysts to quickly determine whether a new binary file is malicious or not, based on a nearest-neighbor search through the database.

Use of graphs is becoming prevalent in depicting structural information. There exist several methods in the database field for indexing and querying graph databases. Most of them focused on exact graph or subgraph matching, i.e., graph or subgraph isomorphism. Ullmann [105] proposed a subgraph isomorphism algorithm based on a tree search approach. However, because both graph and subgraph isomorphism are NP problems [35] (and subgraph isomorphism is proven to be NP-complete), existing algorithms for graph and subgraph isomorphism are prohibitively expensive to use for querying large graphs against a graph database with a large number of graphs. To reduce the search space, several indexing techniques have been proposed using frequent features, including GraphGrep [91], GIndex [115], Tree+Δ [125] and TALE [100], which use paths, graphs, trees and important nodes, respectively, as the main frequent feature to remove graphs that do not match the query. Subgraph isomorphism is then used to prune false positives from the answer set. Several disadvantages of these approaches make them unsuitable for a malware database that contains hundreds of thousands large graphs. First, some of them rely on expensive isomorphism algorithms and thus are only applicable to small graphs. Second, these approaches require all the indexing features to be matched exactly with the query and thus, cannot effectively capture the similarity among malware variants. For example, malware writers often create malware variants by adding new features (e.g., logging) or some cosmetic changes without affecting the essence of the original malware. However, a new variant created this way will not be isomorphic to the original one even though they are similar.

In this work, we take an approximate graph-matching approach and index the malware graph database using graph similarity. Recently, several indexing methods for similarity queries have also been proposed [44, 116]. Most of them are still

built upon exact subgraph isomorphism and therefore, only apply to relatively small graphs, allowing limited approximation. Another widely-used graph similarity metric is the graph-edit distance, which has shown to be suitable for many error-tolerant graph-matching applications [75]. However, because computing graph-edit distance is NP-hard [124], using exact graph-edit distance is feasible only for small graphs. To reduce the computational cost, several methods have been proposed to calculate approximate edit distance [52, 74, 89]. Justice and Alfred [52] proposed a linear programming method for computing graph edit distance, which can be used to derive lower and upper bounds for the exact edit distance. Riesen *et al.* [89] developed a polynomial-time algorithm to compute approximate graph-edit distance using Bipartite Graph Matching. `SMIT` adopts this approach and tailors it to measure distances between malware call graphs. To support similarity queries (e.g., K Nearest Neighbor query), several techniques for metric space search have also been developed. Yianilos [121] proposed the original Vantage Point Tree (VPT) structure for multi-dimensional nearest-neighbor search. Later, several extensions to VPT have been made to improve its efficiency, such as Multi-way VPT [17], Excluded Middle Vantage Point Forest [120], Optimistic VPT [21], and M-tree [123]. They have been successfully applied to various applications, for example, content-based retrieval on multimedia data repositories [15].

Function-call and control-flow graphs have also been used frequently for malware analysis. Carrera and Erdélyi [20] applied graph theory to function-call graphs for clustering existing malware files. Kruegel *et al.* [60] constructed control-flow graphs from network streams and detected polymorphic worms by identifying structural similarities. Briones and Gomez [18] combined function-call graphs, control-flow graphs and entropy of data blocks to automatically classify malware samples. `SMIT` differs from others in that it proposes a function-call graph indexing approach towards the important problem of malware classification. It focuses on developing an efficient

indexing structure to organize and query large malware databases. In addition, SMIT utilizes a graph similarity metric based on an optimal bipartite matching algorithm which can better capture the internal structure of the call graphs.

2.3 Function-Call Graph Extraction

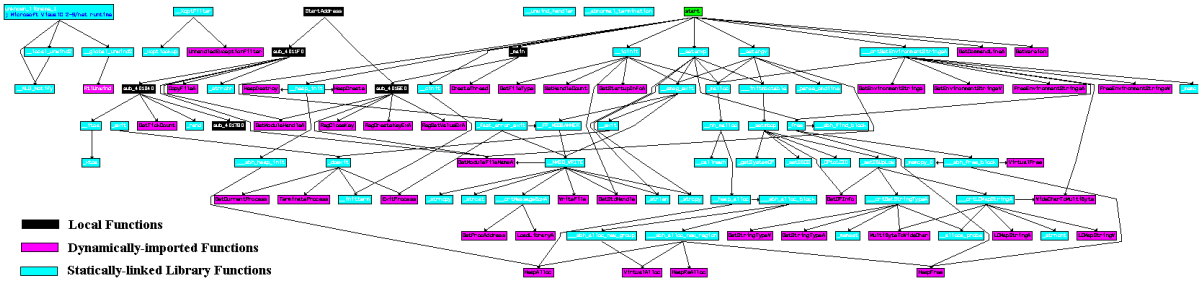


Figure 2.1: The function-call graph of the malware sample Worm.Win32.Deborm.p. Different colors are used to represent different types of functions.

A binary program's function-call graph is a directed graph consisting of a set of vertices (corresponding to functions), a set of directed edges (corresponding to caller–callee relationships) and a set of labels, one for each vertex (containing the attributes of the associated function). Figure 2.1 shows the function-call graph of a malware sample, Worm.Win32.Deborm.p. To facilitate matching between function-call graphs, we classify a program's functions into three categories, represented as different colors in Figure 2.1.

- **Local functions** (black nodes) are functions written by malware writers and usually shared only by malware variants within the same family.
- **Statically-linked library functions** (cyan-colored nodes) are library functions that are statically linked into the final distributed binary, such as Libc, MFC, Delphi Visual Component Library, etc. They tend to be shared by malware samples from different families.

- **Dynamically-imported functions** (pink nodes) are dynamically-linked library (DLL) functions that are linked at run-/load-time, e.g., library functions in Kernel32.dll, User32.dll, advapi32.dll, etc. Since these functions are dynamically linked, their bodies do not appear in malware samples. These functions also tend to be shared across malware families.

Given an incoming malware sample, `SMIT` extracts its function-call graph as follows. First, `SMIT` uses PEiD [51] and TrID [83] to check if the malware file is packed. If so, `SMIT` applies SymPack (an unpacker developed inside Symantec) to unpack or decrypt the malware file. To handle multi-layer packing, `SMIT` applies this step recursively until the file is completely unpacked. Then, `SMIT` uses the popular disassembler IDA Pro [45] to disassemble the malware into an assembly code representation and identify the function boundaries. It then labels each identified function with a symbolic name. For dynamically-imported functions, their names can be found by parsing the IAT (Import Address Table) in the PE header [82] of the malware file. For statically-linked library functions, e.g., strcmp and iota, `SMIT` utilizes IDA Pro's FLIRT (Fast Library Identification and Recognition Technology) [48] to recognize their original names. Because the import and library functions are standard routines, their names are consistent throughout all the programs. However, for local functions, since most malware samples do not come with their symbol tables, their names are in general unavailable. As a result, we assign all local functions with the same name (`sub_`) whenever their true symbolic names are unavailable in the input binary. These local functions will later be matched based on their mnemonic sequences or call-graph structures.

To facilitate matching of local functions, `SMIT` extracts from each local function the sequence of call instructions it contains, and a mnemonic or opcode sequence from instructions in its body. For example, "mov" is the mnemonic for the instruction "mov eax, [0x403FBB]". Such mnemonic sequences are more robust than instruction

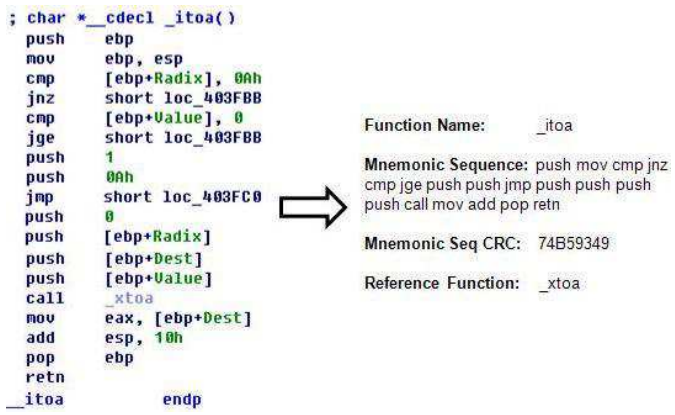


Figure 2.2: Example of a function being represented by a mnemonic sequence and other features.

sequences because they ignore offsets that may change due to code relocation. They are used in the graph-similarity computation as a coarse-grained filter to identify functions from two programs that are likely to be matched. That is, if two functions have similar mnemonic sequences, they are likely to be the same function. `SMIT` also computes the CRC of mnemonic sequences to speed up the exact matching between sequences. With all the information collected from each function (shown in Figure 2.2), `SMIT` is able to construct the call graph representation for each malware file. More formally, `SMIT` defines a program's function-call graph as follows.

Definition II.1. (Function-Call Graph): A function-call graph g is a directed graph defined by 4 tuples $g = (V_g, E_g, \mathcal{L}g, L_g)$, where V_g is the finite set of vertices, each corresponding to a function; $E_g \subseteq V_g \times V_g$ is the set of directed edges where an edge from f_1 to f_2 implies that f_1 contains a function call to f_2, but not vice versa; $\mathcal{L}_g$ is the set of labels each of which is comprised of 3 elements: symbolic function name, mnemonic sequence and CRC value of the mnemonic sequence; $L_g : V_g \rightarrow \mathcal{L}_g$ is the labeling function that assigns labels to vertices.

2.4 Graph-Similarity Metric

Unlike many other graph database management systems, the central component of `SMIT` is a graph database engine that finds the nearest neighbors of a given query graph in a graph database. Rather than its subgraphs or supergraphs, `SMIT` uses a graph-similarity metric that aims to capture the similarity among variants within the same malware family, and that can be computed at low cost. Here we give details of this metric: an approximate graph edit distance.

2.4.1 Graph Edit Distance

Mathematically, a *metric* between elements of a set X is the *distance function* $d : X \times X \to \mathbb{R}$ that satisfies the following properties: *non-negativity*, *identity*, *symmetry* and *triangular inequality*. When applied to graphs, the notion of equivalence is defined in terms of isomorphism—if two graphs are isomorphic, then they are viewed as the same graph. The triangular inequality plays an essential role in the development of indexing schemes because it enables pruning of irrelevant portions of the database.

The edit distance between two graphs measures their similarity in terms of the number of edit operations required to transform one graph to the other. For the purpose of identifying malware variants, the graph-edit distance effectively captures the amount of effort needed to convert one program to another at the function-call graph level, and thus forms an intuitively appealing metric. Given any two graphs, we define the following two elementary operations to transform one graph to another.

1. **Vertex-edit operations** including: σ_R, relabel a vertex; σ_{IV}, insert an isolated vertex; and σ_{RV}, remove an isolated vertex.

2. **Edge-edit operations** including: σ_{IE}, insert an edge and σ_{RE}, remove an edge.

An edit path $P_{g,h}$ between graphs g and h is defined as a sequence $(\sigma_1, \sigma_2, \ldots, \sigma_n)$

of elementary operations such that $h = \sigma_n(\sigma_{n-1}$
$(\ldots \sigma_1(g) \ldots))$. To quantify this similarity, a cost is assigned to each edit operation: $c : \sigma_R, \sigma_{IV}, \sigma_{RV}, \sigma_{IE}, \sigma_{RE} \rightarrow \mathbb{R}$. Then, the cost of an edit path is the sum of the costs of all the constituent edit operations, i.e., $P = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ as $c(P) = \Sigma_{i=1}^{n} c(\sigma_i)$. The *edit distance* between two graphs is defined as the minimum cost of all edit paths between them, i.e., $ed(g, h) = min\ c(P_{g,h})$. If all costs are strictly positive, with insertion cost equal to removal cost, then the graph edit distance satisfies all the mathematical properties associated with a metric. Furthermore, for any graph q, the sum of the distances $ed(g, q)$ and $ed(q, h)$ is the cost of the edit path transforming g first into q and then into h, which is itself an edit path from g to h. Hence, by the minimality of edit distance over all edit paths, the triangle inequality $ed(g, h) \leq ed(g, q) + eg(q, h)$ is maintained. Therefore edit distance is a metric on the space of graphs. In `SMIT`, we assign a *unit cost* to each edit operation.

2.4.2 Approximating Graph-Edit Distance Using Graph Matching

The main drawback of graph-edit distance is its computational complexity, which is exponential in the number of nodes of the graphs. Thus, application of graph-edit distance is feasible only for relatively small graphs, say those with fewer than 50 nodes. Because the number of nodes in malware graphs is significantly larger, we develop heuristic algorithms that can closely approximate the ideal graph-edit distance using graph matching techniques. To this end, we first define the notion of graph matching which is a relaxed notion of correspondence between two graphs used later to calculate the graph edit distance.

To match two unequal-size graphs g and h, we extend the vertex set of each graph as: $V_g^* = V_g \bigcup \epsilon_g$ and $V_h^* = V_h \bigcup \epsilon_h$, where ϵ_g and ϵ_h are sets of dummy nodes created to account for insertions and deletions. In other words, a match from $u \in V_g$ to a dummy node implies the deletion of u from graph g. Similarly, insertion is

denoted by matching a dummy node to $v \in V_h$. Hence, if $|V_g| = m$ and $|V_h| = n$, we take $|\epsilon_g| = n$ and $|\epsilon_h| = m$, We set $|\epsilon_g| = |V_h|$ and $|\epsilon_h| = |V_g|$ so that the extended graph has the same number of nodes. We denote the extended graph for g as $g^* = (V_g^*, E_g, \mathcal{L}_g, L_g \cup \{\epsilon_g\})$ and define the graph matching as:

Definition II.2. (Graph Matching) A matching between two graphs g and h is a bijective function $\phi()$ between two vertex sets, $\phi : V_g^* \rightarrow h_g^*$ such that $\forall v \in V_g^*, \phi(v) \in V_h^*$.

Given a graph matching ϕ between two graphs g and h, the distance (edit cost) between them can be computed by considering mismatched nodes and edges with the following algorithm.

1. Let C_E represent the number of edges that are mapped from one graph to the other. Specifically, for any edge $(i, j) \in E_g$, if $(\phi(i), \phi(j)) \in E_h$, then the matching preserves the edge (i, j) and the counter C_E is incremented by 1.

2. $EdgeCost = (|E_g| - C_E) \times c(\sigma_{RE}) + (|E_h| - C_E) \times c(\sigma_{IE})$. Since we assign unit cost to each edit operation, $EdgeCost = |E_g| + |E_h| - 2 \times C_E$.

3. For any node in graph g that is matched to a dummy node in h, we add $c(\sigma_{RV})$ to the $NodeCost$ to penalize for deleting the node. Similarly, when a node in graph h is matched with a dummy node in g, we add $c(\sigma_{IV})$ to the $NodeCost$.

4. For any two matched nodes, we add $c(\sigma_R)$ to the $NodeCost$ if they have different labels, i.e., the relabeling cost.

5. Edit distance under ϕ is: $ed_\phi(g, h) = NodeCost + EdgeCost$.

Because graph-edit distance is defined as the minimum edit cost between two graphs, the above algorithm casts the problem of computing graph-edit distance into finding a function ϕ that minimizes the total matching cost, i.e., a minimum-cost

bipartite matching problem, where each of the two sides of the bipartite graph corresponds to nodes from one of the two input graphs. whose edit distance is to be computed. An optimal (minimum-cost) bipartite matching can be found in polynomial time ($O(n^3)$) by using the well-known *Hungarian algorithm* [62]. `SMIT` uses an well-known optimal bipartite matching algorithm called the to solve this problem. To further reduce the performance overhead of the Hungarian algorithm, `SMIT` employs various optimizations that exploit properties of the malware programs underlying their function-call graphs. These optimizations are discussed next.

2.4.3 Optimizations

2.4.3.1 Exploiting Instruction-Level Information

Since the complexity of the Hungarian algorithm depends on the number of nodes in the input graphs, the first optimization aims to reduce the number of nodes in the two input graphs that need to be matched by removing those nodes that can be matched through other cheaper means. Specifically, `SMIT` uses each function's mnemonic sequence, CRC value of its mnemonic sequence and symbolic name to quickly determine if a function in one input graph matches another function in the other input graph, and compute a common function set $C = \{v : v \in V_g \bigcap V_h\}$ containing:

- Functions that IDA Pro identifies as static library functions or dynamically-imported functions and that share the same symbolic names in two input graphs.
- Functions that have the same mnemonic sequence and thus the same CRC value of their mnemonic sequence; and
- Functions that have similar mnemonic sequences. We compute the edit distance between the mnemonic sequences of two functions, and consider them a match when the distance is below 15% of the length of the shorter mnemonic

sequence of the two, where the threshold 15% is chosen empirically. We use a greedy algorithm to find all matched functions. That is, we start with two functions that have the smallest edit distance; if their distance is smaller than the threshold, they are marked as a match and put into C. Then, we repeat the same procedure with respect to the remaining functions until no function pair whose edit distance is smaller than the threshold exists.

To further decrease the number of nodes to which the Hungarian algorithm needs to be applied, we apply a neighborhood-driven algorithm [20] that exploits the matched neighbor information associated with functions. Let's call the functions in $C = \{v : v \in V_g \bigcap V_h\}$ *atomic functions* and let $V_g^r = V_g - C$ and $V_h^r = V_h - C$ denote the sets of the remaining functions in g and h that are not yet matched. A *call-sequence signature* for each remaining function is a sequence of calls to atomic functions in this function. If the call-sequence signatures of two functions $f_1 \in V_g^r$ and $f_2 \in V_h^r$ are identical, meaning that they call the same sequence of functions, we generate a match between f_1 and f_2 because they are likely very similar or the same. Whenever a new match between two local functions is found, we move them from V_g^r and V_h^r to the common function set C, and repeat the algorithm until it yields no additional matches. At the end of the process we apply the Hungarian algorithm to the remaining V_g^r and V_h^r. For malware variants from the same family, this optimization can match over 90% of functions. On the other hand, the number of matched functions for malware from different families is often below 20, most of which are shared library functions.

2.4.3.2 Bipartite Graph Matching

The problem of finding a min-cost bipartite graph matching can be solved in polynomial-time using the Hungarian algorithm [62]. Once the lowest-cost match is found, it can be used to create an edit path and compute an estimate of the true edit

distance (Section 2.4.2). Note that, although the Hungarian algorithm is optimal, the edit-distance result returned by the match function ϕ that the algorithm finds is only suboptimal [62], because the *cost matrix* used to search for the optimal node assignment is computed without global knowledge (to be elaborated). To mitigate this problem, we develop an optimized Hungarian algorithm that biases the matching process towards the neighboring functions of already-matched functions. Comparing with the original algorithm, the improved Hungarian algorithm often finds a better matching function ϕ that yields closer approximation to the true edit distance.

The algorithm first constructs a complete bipartite graph with vertex classes $X = V_g^r \bigcup \epsilon_g$ and $Y = V_h^r \bigcup \epsilon_h$, where ϵ_g and ϵ_h are sets of dummy nodes with $|\epsilon_g| = |V_h^r|$ and $|\epsilon_h| = |V_g^r|$. In this bipartite graph, each edge is assigned a weight corresponding to an estimate of the cost of mapping a vertex $x \in X$ to a vertex $y \in Y$. The choice of weights for the edges of the bipartite graph is a vital component of the algorithm, as well-assigned weights that are closer to the real cost will result in a near-optimal edit path, and thus, the Hungarian estimate will more closely approximate the true edit distance. Assume the first graph g_r has size n, and the second graph h_r has size m, we form an $(m+n) \times (m+n)$ **cost matrix**. In the top left we have an $n \times m$ sub-matrix giving the costs of matching a real node in g to a real node in h. In the bottom right is an $m \times n$ zero sub-matrix, representing the costs of associating a dummy node with another dummy node. Finally, the off-diagonal square sub-matrices give the cost of pairing a real node from a graph to a dummy node from the other graph (thereby deleting it). On the diagonal, these matrices store the cost of deleting a node and all its incident edges (both In and Out). We set all non-diagonal components of these matrices to ∞ to ensure that each real node is associated with a unique dummy node. This will simplify the matching process.

In [89], the cost of matching any two real nodes was taken simply as the relabeling cost. To find a better estimate of the true edit cost, we improve the algorithm by

considering the edges as well. Specifically, the cost estimate, $C_{i,j}$, of matching node i to node j, is the sum of the **Relabeling Cost** and the **Neighborhood Cost**, where the latter is calculated from the difference between i and j's adjacent nodes. This introduces structural information by giving a lower-bound for the edit cost of matching the neighbors of i and j.

1. Relabeling Cost: If the label of node i is not the same as the label of node j, we set $C_{i,j}$ to be the relabeling cost (σ_R).

2. Outgoing Neighborhood Cost: For any graph g and node $i \in V_g$, $N^g_{Out}(i) \equiv \{L_g(k)|(i,k)\} \in E_g$. Then, the outgoing neighborhood cost of matching node i to node $j \in V_h$ is $|N^g_{Out}(i)| + |N^h_{Out}(j)| - 2 \times |N^g_{Out}(i) \cap N^h_{Out}(j)|$ to $C_{i,j}$.

3. Incoming Neighborhood Cost is similarly defined with the incoming edges.

The cost computed from the above algorithm is a lower-bound of the true edit cost for the following reasons. As mentioned in Section 2.3, due to lack of symbolic information for all local functions written by malware writers, we assign the same label to those functions. As a result, when computing the estimated matching cost between i and j, any local functions in i's and j's neighborhood are conservatively considered matched (i.e., incurring no matching cost). However, in the final matching function ϕ (found by applying the Hungarian algorithm on the cost matrix), these two neighbor nodes can be unrelated, in which case, the true edit cost between i and j is higher than the estimate. In other words, because the cost matrix is predetermined, the algorithm will only be able to consider the local structure of the nodes without any information about the matching. This lack of global knowledge when computing the cost matrix leads to the sub-optimality of the resulting edit distance as calculated by the algorithm in Section 2.4.2, even though the Hungarian algorithm by itself is optimal in the sense that it finds the min-cost matching according to the pre-determined cost matrix. To alleviate this problem, in the next subsection, we present

our improved Hungarian algorithm that actively exploits the structural information of already-matched nodes as the algorithm progresses.

2.4.3.3 Neighbor-Biased Hungarian Algorithm

One drawback of the standard bipartite matching approach to computing the graph-edit distance is that it assumes a fixed cost of matching two function nodes. However, as observed in [44], when two nodes are matched, their neighbors are also likely matched, because if more neighbors of a node are matched with those of another node, the edge-edit cost of matching these two nodes will decrease (thus reducing the real edit cost). Based on this intuition, we develop a modified Hungarian algorithm that adaptively biases the order of matching towards those pairs of nodes whose neighboring nodes have already been matched.

Given two malware call graphs g and h, we first find the initial set of matched functions (Section 2.4.3.1). For each matched function f, we decrease the cost (in the cost matrix) of matching all the unmatched neighbors of function f in g with their counterparts in h by a predefined percentage. Then, the Hungarian algorithm is applied to the remaining graphs g_r and h_r with the updated cost matrix. In each iteration of the algorithm, whenever two functions, for instance (u, v), are chosen to be matched, the costs of matching their unmatched neighbors in the cost matrix are similarly lowered, thus increasing their chances of being matched later by the algorithm. The procedure repeats itself until a complete match is found in the bipartite graph. As an additional optimization, whenever (u, v) is selected to be matched, the amount of cost reduction for their unmatched neighboring functions is positively proportional to the matching quality of (u, v), defined as the percentage difference between the mnemonic sequences of (u, v). Intuitively, the extent to which the Hungarian algorithm is biased toward the neighbors of a matched node pair is proportional to the degree to which they are considered matched. The pseudocode for the new algorithm

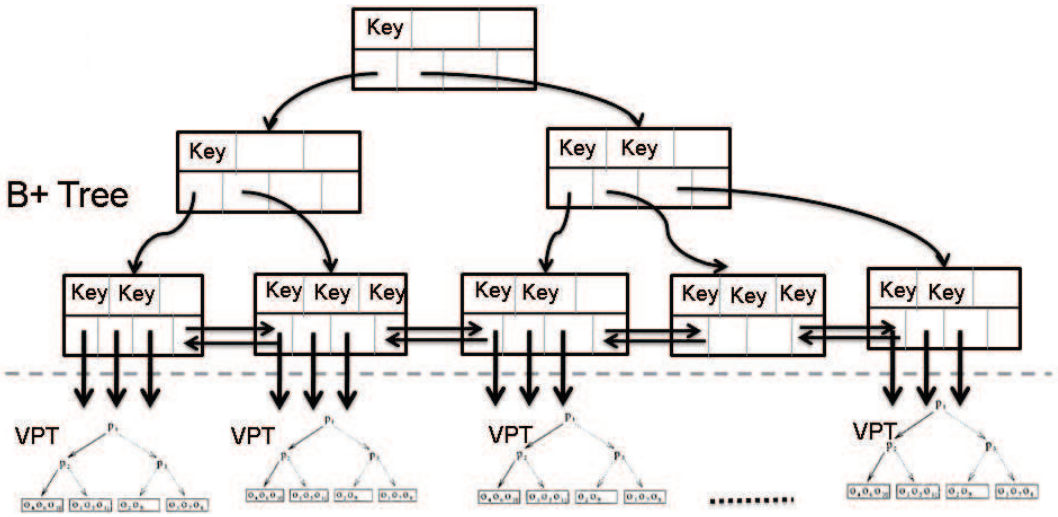


Figure 2.3: Multi-resolution indexing structure.

is shown in Algorithm1.

The above algorithm generates the cost-minimizing matching between function nodes $\phi : V_g \cup \epsilon_g \rightarrow V_h \cup \epsilon_h$, from which the edit-path cost (denoted as $ed_\phi(g, h)$ under ϕ) can be calculated, which is a close approximation to the true edit distance. Note that $ed_\phi(g, h)$ gives the cost of a particular edit path from g to h. The minimality of edit distance across all edit paths ensures that the distance from the Hungarian method is an upper bound on the edit distance. That is, for any two graphs g and h, $ed(g, h) \leq ed_\phi(g, h)$.

2.5 Multi-Resolution Indexing

Having defined a similarity metric, the next important problem is to efficiently index our graph database such that when a new query comes along, the graphs most similar to it can be retrieved with as few distance computations as possible. In this section, we introduce our multi-resolution indexing scheme to achieve this goal.

Algorithm 1 NBHA: Neighbor-Biased Hungarian Algorithm

Input: A bipartite graph with vertex classes $X = V_g^r \bigcup \epsilon_g$ and $Y = V_h^r \bigcup \epsilon_h$. and a Cost Matrix C
Output: the minimum-cost node Matching M
Create a weight matrix ω where $\omega_{i,j} = \max\{C_{i,j}\} - C_{i,j}$ so that the problem is converted to the maximum-weight matching in bipartite graphs based on the weight matrix ω
STEP 1:
Find an initial feasible vertex labeling function $\ell : V \rightarrow \mathcal{R}$ such that $\ell(x)+\ell(y) \geq \omega(x, y)$ and an initial matching M in the Equality Graph $G = (V, E_\ell)$ where $E_\ell = \{(x, y) : \ell(x) + \ell(y) = \omega(x, y)\}$.
STEP 2:
if M is perfect i.e., every vertex is adjacent to some edge in M **then**
  GOTO DONE
else
  Pick a free vertex $u \in X$ and set $S = \{u\}, T = \emptyset$
end if
STEP 3:
Define neighbor of $u \in V$ and a set $S \subseteq V$ to be $N_\ell(u) = \{v : (u, v) \in E_\ell\}$ and $N_\ell(S) = \bigcup_{u \in S} N_\ell(u)$
if $N_\ell(S) = T$ **then**
  $\alpha_\ell = min_{s \in S, y \notin T}\{\ell(x) + \ell(y) - \omega(x, y)\}$,

$$\ell(v) = \begin{cases} \ell(v) - \alpha_\ell & \text{if } v \in S \\ \ell(v) + \alpha_\ell & \text{if } v \in T \\ \ell(v) & \text{otherwise.} \end{cases}$$

else
  Pick $y \in N_\ell(S) - T$
  if y is matched to some vertex say z **then**
    $S = S \bigcup \{z\}$, $T = T \bigcup \{y\}$
    GOTO STEP 3
  else
    Define $slackx[y]$ to be the vertex that $\ell(slackx[y])+\ell(y)-\omega(slackx[y], y) = slack[y]$ where $slack[y] = \min_{x \in S}(\ell(x) + \ell(y) - \omega(x, y)$.
    Define $prev[cx]$ to be the parent vertex of cx in the alternating path
    for $cx =$ slackx$[y]$, $cy = y$, ty; $cx \geq 0$; $cx = prev[cx]$, $cy = ty$ **do**
      $ty =$ col_mate$[cx]$
      row_mate$[cy] = cx$ {Augment M}
      col_mate$[cx] = cy$
      for each $i \in$ neighbors of cx and $j \in$ neighbors of cy **do**
        increase $\omega(i, j)$ {Bias towards the matching between cx's neighbors and cy's neighbors}
      end for
    end for
    GOTO STEP 2
  end if
end if
DONE: Matching pairs are indicated by two arrays: row_mate and col_mate

2.5.1 Overview

For the purpose of identifying malware variants, it is not necessary to pinpoint the exact nearest neighbor for a new malware file. As long as one can identify a neighbor that is close enough to the new file, one can "convict" it. For scalability to a large database, `SMIT` exploits this latitude and incorporates a multi-resolution indexing technique that makes a good balance between pruning efficiency and search effectiveness.

Conventional indexing methods decompose a database into partitions and organize them hierarchically, so that a search can focus on a subset of these partitions at each level of the hierarchy, thus reducing the total number of database items that it needs to touch. These indexing methods are inadequate for `SMIT` for two reasons. First, `SMIT` requires an indexing scheme that supports nearest-neighbor search, rather than exact search that conventional methods are designed for. Second, since computation of graph similarity is expensive, `SMIT` must minimize the number of such computations. For instance, our evaluation shows that a modern desktop PC can perform an average of 20 graph-similarity computations per second for our malware set. At this performance level, even if an indexing scheme could reduce the number of graphs that a search needs to touch, to less than 10% of the database, it will still take hours to answer a single query for a database of 1,000,000 malware graphs.

To address the first problem, `SMIT` organizes the input malware graph database using the optimistic Vantage Point Tree (VPT), which is designed for nearest-neighbor search and can exploit the fact that sufficiently near neighbors are usually good enough. To solve the second problem, `SMIT` uses a two-level indexing scheme, where the first level is a standard B+-tree index based on coarse-grained malware features that can be computed inexpensively and that can effectively prune irrelevant parts of the malware database. Graphs associated within each leaf node of the B+-tree index are organized with a second-level index, i.e., the VPT Tree, which uses a more

accurate but computationally more expensive graph-similarity function to pinpoint the most similar neighbors. The two-level indexing (Figure 2.3) in `SMIT` is an instance of multi-resolution indexing because similarity functions with different accuracy and computational requirements are used in the different levels of the index tree.

2.5.2 B+-tree Index Based on Malware Features

The feature vector used in `SMIT`'s first-level index must satisfy two requirements. First, its computation cost must be low. Second, it must be able to identify parts of the malware database that are not relevant to a given malware query. That is, the feature vector needs to be able to pinpoint the obviously irrelevant, but not necessarily the most relevant. Specifically, `SMIT` uses the following feature vector $v = (N_i, N_f, N_x, N_m)$ derived from the assembly code of each malware program, where:

- N_i: total number of instructions,
- N_f: total number of functions,
- N_x: total number of control transfer instructions, such as jumps and calls, which indicates the degree to which a program deviates from a straight-line code and thus is a good approximation of a program's complexity, and
- N_m: median number of instructions per function.

The feature vector has the following property: if two malware programs are similar to each other, so are their feature vectors. However, if two malware are dissimilar, their feature vectors may or may not be similar. Therefore, it is only useful when the feature vectors of two malware are drastically different, meaning that the underlying programs are definitely different, but not when their feature vectors are somewhat different or similar.

Because leaf nodes in a B+tree need to be ordered by their keys (feature vectors), we impose a total ordering among feature vectors by giving priority to more useful features ($N_i > N_f > N_x > N_m$). We also augment the B+ tree structure by adding a *backward sibling pointer* to each leaf node, which points to the previous leaf node. Together with the *forward sibling pointer* in the B+-tree, it facilitates navigation across leaf nodes and indexed search.

Given a malware query, `SMIT` first extracts its feature vector and uses it as a key to search the B+-tree index. Suppose the probing ends in a lead node X. `SMIT` then follows X's forward and backward sibling pointers to locate N leaf nodes before and after X, and further explores the second-level index trees (VPT) associated with these $2N+1$ leaf nodes. Here N is an empirically-determined parameter that is designed to reduce the probability of the feature vector pruning away sufficiently close neighbors. Because these $2N + 1$ VPTs are independent of one another, they can be queried in parallel to reduce the query response time. Finally, the K nearest neighbors returned from the exploration of each of the $2N+1$ VPTs are combined to determine the final K nearest neighbors.

2.5.3 Optimistic Vantage Point Tree

The Vantage Point Tree (VPT) is designed for database items whose similarity to each other must be explicitly computed (e.g., graphs), and exploits the triangular inequality to prune irrelevant database items. To construct a VPT for a graph database, we first select a graph as the root pivot V, compute the distance between V and all the remaining graphs, and then divide these graphs into M approximately equal-sized partitions ($P_i, i = 1, 2, ..., m$) based on their distance to V. Geometrically, this method first places all graphs essentially split the area around the pivot into m *concentric areas*, each corresponding to a child of the current node. In addition, at the pivot V, we record the distance range associated with each partition P_i, which is

represented by $low[i]$ and $high[i]$. This same procedure is repeated for each partition recursively, until all partitions fall below a certain size. The construction of the index takes $O(n^2)$ time in the worst case where n is size of the database. In `SMIT`, n is equal to the number of elements in the leaf node of the higher level B+-tree structure.

VPT supports two types of search: *range* and *K-nearest-neighbor* (KNN) search. `SMIT` focuses on KNN search because malware analysts are more interested in locating all existing malware samples that are most similar to a new sample. In addition, it is quite hard for user to specify a meaningful range value. As a result, here we only present the k-NN search algorithm. Given a query graph g, the K-nearest-neighbor (KNN) search of a VPT with a root pivot p starts with computation of the edit distance $d(p, q)$ between p and q, and then decide which partitions to explore further by exploiting the triangular inequality of the distance metric. More specifically, let δ_{now} be a parameter indicating to the search algorithm that it should ignore any database item whose distance to the query q is larger than δ_{now}. Given δ_{now}, the search only needs to explore those partitions whose distance range overlaps with the range of interest, $[d(p, q) - \delta_{now}, d(p, q) + \delta_{now}]$, as shown in Figure 2.4. That is, partition P_i is pruned if and only if

$$high[i] < d(p, q) - \delta_{now} \text{ or } low[i] > d(p, q) + \delta_{now}. \tag{2.1}$$

This search procedure is applied recursively at each visited node until all nodes are either pruned or visited.

Eq. (2.1) shows that at each node, the pruning power of the VPT search algorithm is dependent on the value assigned to δ_{now}. If δ_{now} is small, only a few partitions need to be traversed. However, too small a δ_{now} may lead to pruning of the partitions that actually contain the nearest neighbors. One way to keep δ_{now} as small as possible is to update it during the search. At any point in a KNN search, the algorithm remembers

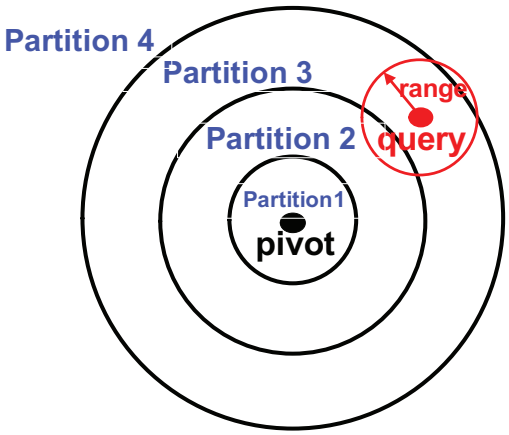


Figure 2.4: Pruning on a VPT based on the triangular inequality

the K closest neighbors that it has encountered so far together with their distance to the query graph q in a priority queue, and sets δ_{now} to the largest of these distance values after accumulating K closest neighbors. Every time the search encounters a database item p whose $d(p, q)$ is smaller than δ_{now}, it adds p together with $d(p, q)$ to the priority queue and updates δ_{now} accordingly. Another way to reduce the value of δ_{now} is to traverse the partitions that are closer to the query graph earlier than those that are farther away. For example, in Figure 2.4, partition 3 is traversed before partition 2 or 4, because closer partitions are more likely to contain closer neighbors.

To make an optimal balance between accuracy and efficiency when initializing δ_{now}, we take an optimistic approach (OVPT) [21] by starting with a small initial δ_{now} value, and exponentially increasing it at subsequent iterations if previous iterations fail to identify K nearest neighbors. Specifically, for a VPT rooted at node p, the initial δ_{now} is chosen to be

$$\delta_{now} = \max_{i=1}^{m-1} \frac{low[i+1] - high[i]}{2} + 1 \tag{2.2}$$

where $low[i]$ and $high[i]$ are the lower and upper ends of the i-th partition's distance range. This choice of the initial δ value guarantees that for any query graph

q, at least one partition will be traversed, because $d(q,p)$ will fall within at least one partition's extended distance range, $[low[i] - \delta_{now}, high[i] + \delta_{now}]$.

When the initialized value of δ_{now} is too small, the search may not find all K nearest neighbors. In such a case, `SMIT` increases the initialized value δ_{now} using $\delta_{now,M} = \delta_{now,M-1} + \alpha$ or $\delta_{now,M} = \delta_{now,0} * \beta^{M-1}$where α and β are additive and multiplicative constants and M is the number of iterations that have been attempted to find the K nearest neighbors. To reduce the performance overhead of OVPT, all the distance-computation results in previous iterations are cached so that no distance computation may ever be done more than once in an OVPT search.

The performance gain of OVPT comes from two sources. First, we notice empirically that there is a big difference between the time needed to locate the K nearest neighbors and the time needed to verify that they are indeed nearest neighbors. Using a smaller initial δ_{now} value significantly reduces the verification cost because it cuts down the number of candidates considered, especially when the query graph is indeed close to its nearest neighbors. Second, the optimistic approach carries almost no additional performance overhead because all distance-computation results in previous iterations are cached and can thus be readily reused. More concretely, any partitions that are not pruned in the $(M-1)$-th iteration will never be pruned in the M-th iteration because $\delta_{now,M-1} < \delta_{now,M}$. This means that all the distance computations in previously iterations are necessary, and their caching guarantees that no distance computation will be done more than once.

2.6 Evaluation

In this section, we apply `SMIT` to a large collection of real-world malware files and evaluate its performance using K-nearest-neighbor (K-NN) search queries based on the following three metrics: *effectiveness* (whether the results produced by `SMIT` are meaningful and similar to those produced by human analysts), *efficiency*, and

scalability. We focus on the K-NN search, because, given the polymorphic nature of modern malware, finding the most similar samples in the database to a given malware file is more useful in determining if it is malicious than pinpointing its exact match or the ones that are sub-graphs/super-graphs of it.

2.6.1 Experiment Setup

The dataset used in the evaluation contains 102,391 unique malware programs recently submitted to Symantec Corporation. These malware samples range from simple trojan/virus (less than 100 instructions) to considerably larger malware (more than hundreds of thousand instructions). All the malware files had been analyzed by human experts and classified into families. Each file is labeled with a VID (Virus ID) representing the malware family to which it belongs. As a result, we can determine that a binary file used in a query is a variant of an existing malware file if both share the same VID (i.e., belong to the same family). In total, these malware programs come from 1747 families. We first create a function-call graph representation for each malware file. The graphs have an average number of 504 nodes and 1074 edges, and a maximum number of 37809 nodes and 83737 edges. We implement `SMIT` in C++ and conduct all experiments on a Dell R905 Server with 1.90 G Quad-Core CPU running Windows Server 2003. `SMIT` is a CPU-bound application because of the graph distance computation and has a moderate memory requirement (less than 100MB).

To evaluate the performance of `SMIT`, we use the following three metrics: 1) the percentage of index entries that are accessed to locate the K nearest neighbors of the query file; 2) the percentage of the returned K-NN malware files that are in the same family as the query file; and 3) the average runtime of K-NN search. The first metric measures the average portion of the `SMIT` index tree that needs to be examined to service a query. The second reflects the accuracy and effectiveness of

Feature	Min	Max	Average	Median	STD
N_i	1	1807413	24233.0	7319	55390.9
N_f	1	37130	480.6	85	1077.6
N_x	1	9998	39.1	18	181.4
N_m	0	731350	4932.3	1090	10519.7

Table 2.1: Statistics of different features in the feature vector

the SMIT index tree in correctly identifying a new malware. The last one represents the total computation cost for each query. Because SMIT comprises two indexing structures (B+tree and OVPT), we first evaluate them separately and then their aggregate performance when they are combined.

2.6.2 Effectiveness of B+-tree Index

The first-level B+-tree index in the SMIT index tree uses a computationally economical feature vector representation to attain pruning-efficiency. The minimum, maximum, average and median value for different features in this feature vector are summarized in Table 2.6.1, showing that the value distribution of different features varies significantly across malware samples.[1] This wide variation gives the feature vector considerable pruning power and enables SMIT to search only a small number of most relevant VPT trees.

SMIT's B+ tree index takes the following two parameters: 1) the fan-out of each B+ tree node (the maximum number of data entries in each node); 2) the number of adjacent leaf nodes (denoted as N) whose associated second-level VPT trees are further searched. As the fan-out parameter increases, more keys and pointers can be packed into a B+ tree node, fewer nodes are required to hold the index, and fewer tree nodes need to be accessed during a query search. However, larger fan-out parameters also require bigger second-level VPT trees to be explored to achieve better accuracy.

[1]There are very low feature values such as 0 or 1, because some malware employ various packing or anti-disassemble techniques and cannot be successfully disassembled.

This is a typical trade-off between query result accuracy and computation overhead. According to our experience, setting the fan-out parameter to between 300 to 400 achieves a good balance between query result accuracy and computation overhead. By default, `SMIT` sets the fan-out of its B+ tree index to 400, which results in a three-level B+ tree with 209 leaf nodes. On average, each leaf node contains 273 keys (the occupancy ratio 68.3%) and 398 malware programs (some are mapped to the same key). 65% of time, malware programs that are mapped to the same key also have the same VID, i.e., belong to the same malware family.

To evaluate the effectiveness of `SMIT`'s B+ tree index, we randomly select 426 unique malware files and use them as queries against the `SMIT`'s malware database. For 90.8% of these queries, the returned B+tree leaf node contains at least one malware sample that belongs to the same family as the query, and for 96.2% of them, the returned leaf node or its immediate two neighboring leaf nodes contain at least one malware sample that belongs to the same family as the query. Although the end-to-end accuracy in pinpointing a query file's nearest neighbor also depends on `SMIT`'s second-level indexing, i.e., OVPT, and is thus smaller, the high success rate of finding samples of the same malware family as the query file in the same or close-by leaf nodes, demonstrates the efficacy of `SMIT`'s choice of feature vector as used in its B+ tree index.

2.6.3 Quality of Graph-Similarity Metric

Accurate graph-distance metric is crucial for `SMIT`'s VPT to correctly prune away irrelevant parts of its malware graph database while servicing K-NN search queries. Therefore, we first evaluate the quality of the proposed graph distance metric—Neighbor Biased Hungarian Algorithm (NBHA). We compare NBHA with the original Hungarian Algorithm (OHA) [89], the Neighbor Biased Matching (NBM) algorithm [44] and a Greedy algorithm, which computes the distance between two graphs

from an edit path formed by repeatedly matching the most similar node pairs according to the cost matrix. The results of all these algorithms, including NBHA, have been shown to be an upper bound for the Exact Graph-Edit Distance (EGED). Because EGED computation incurs an exponential cost, we cannot directly compare NBHA with EGED. Instead, we qualitatively evaluate the closeness of NBHA to EGED by computing a graph distance metric called the *multi-set degree-vector distance* (MSDV), which compares the vertices' label and in/out degree between two graphs without considering their connectivity structure. It has been shown that the MSDV distance is a lower bound for the exact edit distance [28].

We randomly select 66 malware graphs, and compute their pair-wise distance using the graph-distance metrics, NBHA, OHA, NBM, Greedy and MSDV. We order the pair-wise distance values obtained from the NBHA algorithm, and present the distance values from other algorithms according to this order. The results are shown in Figure 2.5, where each point on the X-axis corresponds to a particular pair of graphs. In general, NBHA is a good approximation to EGED. By definition, true edit distance (EGED) lies between its upper-bound metrics (NBHA, OHA, NBM, Greedy) and lower-bound metric (MSDV). Because in many cases the upper bounds and lower bound shown in Figure 2.5 are close to each other, these bounds empirically approximate EGED effectively. Moreover, NBHA outperforms other upper-bound metrics (OHA, NBM and Greedy algorithm) in terms of accuracy, because in most cases NBHA's results are smaller than other algorithms'. For upper-bound metrics, smaller metric values imply more accurate approximation to EGED. Specifically, NBHA results are smaller than or equal to those of OHA and NBM, about 95% and 70% of all graph-distance computations in this experiment, respectively.

Next, we evaluate the accuracy and effectiveness of NBHA in terms of the similarity of NBHA results to those produced by human analysts. Specifically, if the distance between two malware files is considered sufficiently small according to NHBA, would

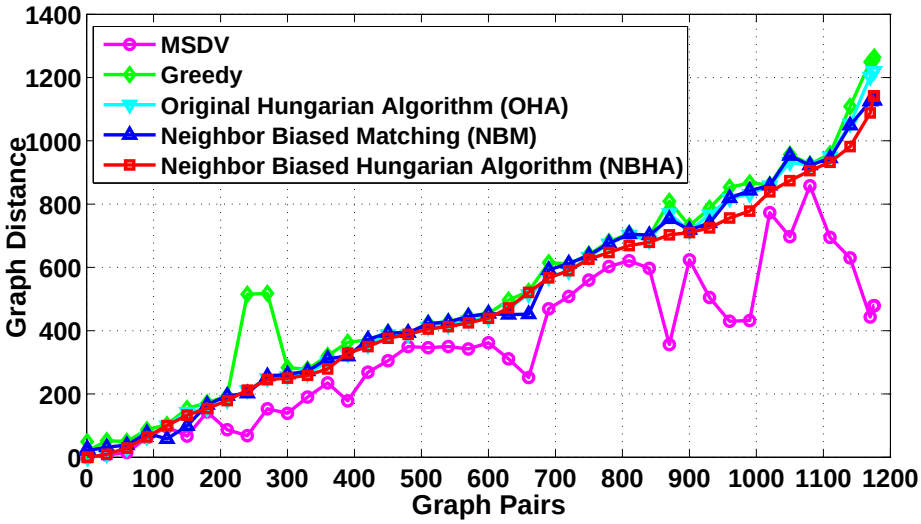


Figure 2.5: Quantitative comparison among graph distance metrics from NBHA, OHA, NBM, Greedy and MSDV. The X-axis corresponds to a sequence of graph pairs.

the human analysts classify them into the same malware family? To answer this question, we randomly selected from the test database 991 malware samples that belong to 122 malware families. In each experimental run, we first select one malware sample as a query and build up a VP Tree for the remaining 990 malware samples. Then, we perform a K-NN search for the query to find the K malware samples that are closest to the query. We repeat the above process for each of the 991 malware samples while varying K, and summarize the results in Table 2.2. In this table, a K-NN query result is a *Success* if at least one out of K nearest neighbors belongs to the same malware family as the query malware file. *Average Hit* is defined as the average number of the returned K nearest neighbors that are in the same family as the query malware. Results in this table suggest that NBHA is effective in classifying unknown malware samples, because it not only achieves high success rate (over 80% for $K >= 5$) but also produces correct labeling in many cases because the most prevalent malware family among the K nearest neighbors is indeed the query malware's family. This result shows that `SMIT` can indeed facilitate, and even automate, the

K=1	K=3		K=5	
Success Rate	Success Rate	Average Hit	Success Rate	Average Hit
71.30%	78.20%	2.36	80.10%	3.11

K=7		K=9	
Success Rate	Average Hit	Success Rate	Average Hit
81.80%	3.64	82.50%	4.14

Table 2.2: Accuracy and effectiveness of the NBHA in terms of K-NN search results

process of convicting incoming malware samples.

2.6.4 Efficiency of Optimistic VPT

We now evaluate the efficiency of Optimistic Vantage-Point Tree (OVPT) using the percentage of index entries (PIE) that need to be accessed to locate the K nearest neighbors of a query file. Because accessing each index entry involves one graph-distance computation, PIE is a proper metric that captures OVPT's computation cost.

We first explore the performance impact of the fan-out factor of `SMIT`'s OVPT (i.e., the number of children each tree node has) and the results are plotted in Figure 2.6. Although a larger fan-out factor reduces the number of levels in the tree, it also increases the number of child nodes that need to be explored at each tree level, because the coverage of each child node is smaller and more of them intersect with the current query range. As a result of these two conflicting influences, Figure 2.6 shows that the fan-out factor does not have a significant impact on PIE. However, the larger fan-out factor increases slightly the overall computation overhead of the OVPT.

Intuitively, as K decreases, less graph-distance computation is required to service each query, because smaller K allows δ_{now} to decrease faster so that fewer partitions of each intermediate OVPT visited need to be traversed. However, in practice, a K

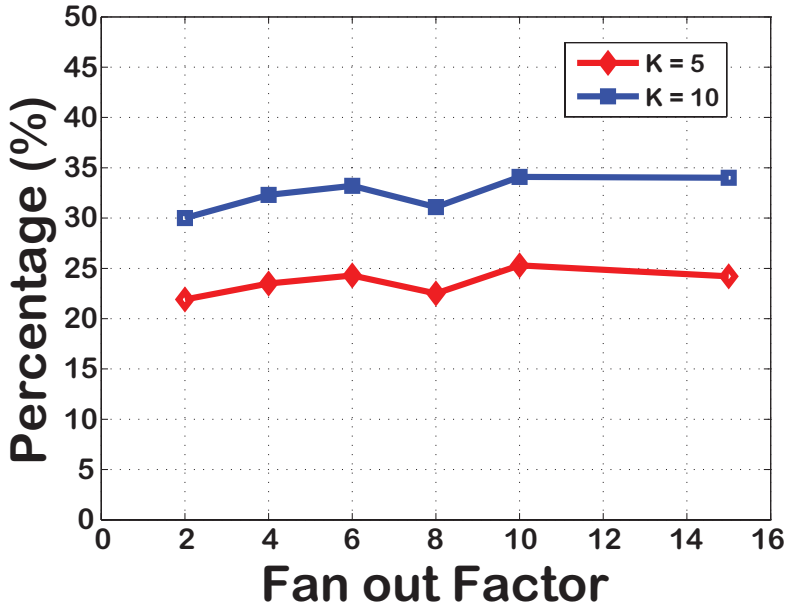


Figure 2.6: Percentage of index entries (PIE) accessed versus the fan-out factor of the VP tree

value of between 5 and 10 is required for human analysts to determine if an incoming binary file is malicious or not. Specifically, if a dominant number of returned K neighbors belong to the same family, there is a very good chance that the query binary file indeed belongs to that family. As shown in Figure 2.7, although PIE increases with K, `SMIT`'s OVPT can still prune away an average of about 70% of the database even when $K = 10$, i.e., for 10-NN search queries. This result demonstrates the effectiveness of `SMIT`'s OVPT index.

Finally, we evaluate the scalability of `SMIT`'s OVPT with respect to the number of graphs being indexed. Because each leaf node in `SMIT`'s first-level B+ tree corresponds to a second-level OVPT tree, this evaluation also helps shed light on the impact of the fan-out factor of the first-level B+ tree. We construct OVPT trees that contain a different number of malware samples, from 100 to 1000 in increments of 50, and for each resulting OVPT, we query it with 100 randomly-selected malware samples and measure the average number of graph distance computations required for different

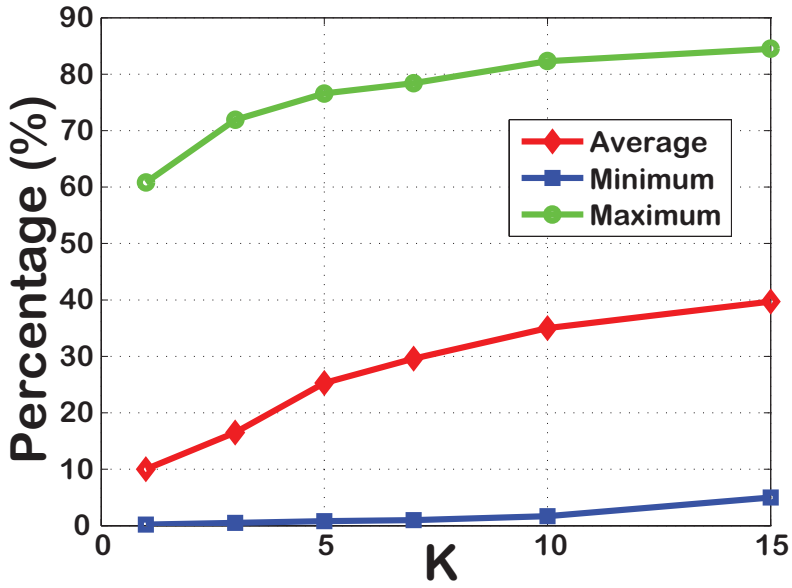


Figure 2.7: PIE vs. the number of nearest neighbors requested (K) (fan-out factor is 10)

values of K. Figure 2.8 summarizes the results and suggests that the number of graph distance computations approximately increase logarithmically with the size of the OVPT tree (the time complexity of searching VP tree is $O(logn)$ [122]), demonstrating its scalability. This also suggests that the number of child nodes explored at each tree level remains largely the same regardless of the total number of levels in the index tree.

2.6.5 Evaluation of Multi-Resolution Indexing

Despite the great pruning power of the OVPT tree, it cannot be directly applied to organize the entire malware graph database, which we envision will grow to millions. For example, even if an OVPT tree can achieve an excellent PIE of 10%, pinpointing the nearest neighbors of a query in a 100,0000-malware database necessitates over 10,000 graph-distance computations, which is unacceptable for practical use. To ensure reasonable response time while maintaining good query accuracy, `SMIT` uses a

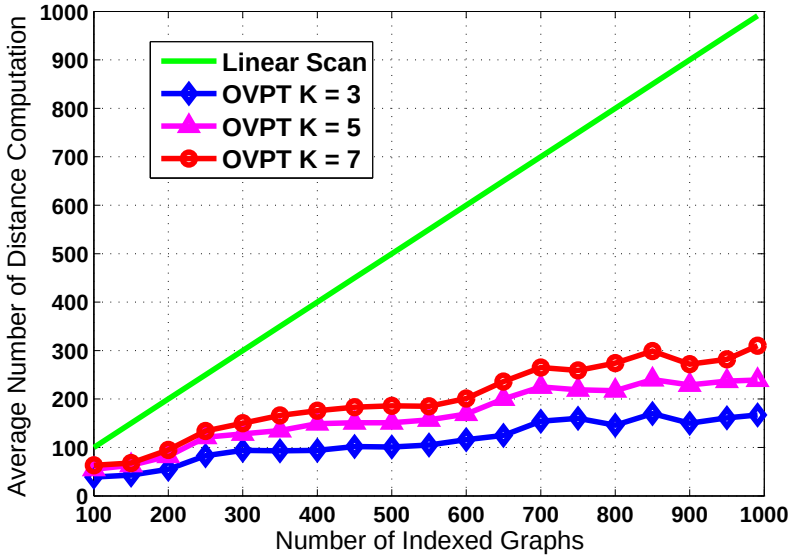


Figure 2.8: Scalability of the VP tree with respect to the number of indexed graphs

multi-resolution indexing structure that removes irrelevant parts of the database with a B+ tree and queries multiple relevant OVPT trees *in parallel*. Next, we evaluate the accuracy and performance of `SMIT`'s combined indexing structure using 102,391 unique malware programs.

2.6.5.1 Impact of N on Query Accuracy

The parameter N for `SMIT`'s B+ tree determines the number of sibling leaf nodes $(2N + 1)$ in the first-level index that need to be searched in the second-level index search. A larger N improves the probability of locating the true K nearest neighbors in the database of the query file, and of correctly identifying the true malware family it belongs to, if any. However, increasing N inevitably increases computational overhead because more second-level OVPT trees are searched. To evaluate the impact of N on `SMIT`'s accuracy, we randomly select 50 malware programs and perform K-NN searches for them with different K (5 and 10) and N (0, 1, 2, 3 and 4). Table 2.3 summarizes the experimental results. *Success Rate* and *Average Hit* are defined

	K=5		
n	Success Rate	Dominant Family Rate	Average Hit
0	76.7%	66.7%	3.24
1	83.3%	70.0%	3.20
2	83.3%	66.7%	3.12
3	86.7%	66.7%	3.13
4	86.7%	66.7%	3.13
	K=10		
n	Success Rate	Dominant Family Rate	Average Hit
0	78.3%	65.2%	6.29
1	87.0%	69.6%	6.30
2	87.0%	69.6%	5.99
3	91.3%	69.9%	5.91
4	91.3%	69.9%	5.98

Table 2.3: Impact of N on the accuracy of identifying the malware family of a query binary file

as in Section 2.6.3 and *Dominant Family Rate* is defined as the percentage of 50 experiments where the most prevalent family among K returned nearest neighbors is also the family to which the query malware belongs. As expected, Success Rate increases with the increase in N. However, the difference in Success Rate among $N = 2, 3$ and 4 does not appear significant enough to warrant the extra performance cost. This is because leaf nodes that are far away from the current leaf node usually contain malware files whose feature vectors are quite different from the query malware, indicating that they are likely not in the same family as the query malware. Hence, exploring more leaf nodes (i.e., larger N) does not significantly improve the accuracy because they are less likely to contain malware with the same family. In our current `SMIT` prototype, we choose $N = 2$ as the default setting. In addition, the high values of Dominant Family Rate and Average Hit in Table 2.3 also demonstrate the effectiveness of `SMIT`'s multi-resolution index in helping human analysts identify the malware family of incoming samples.

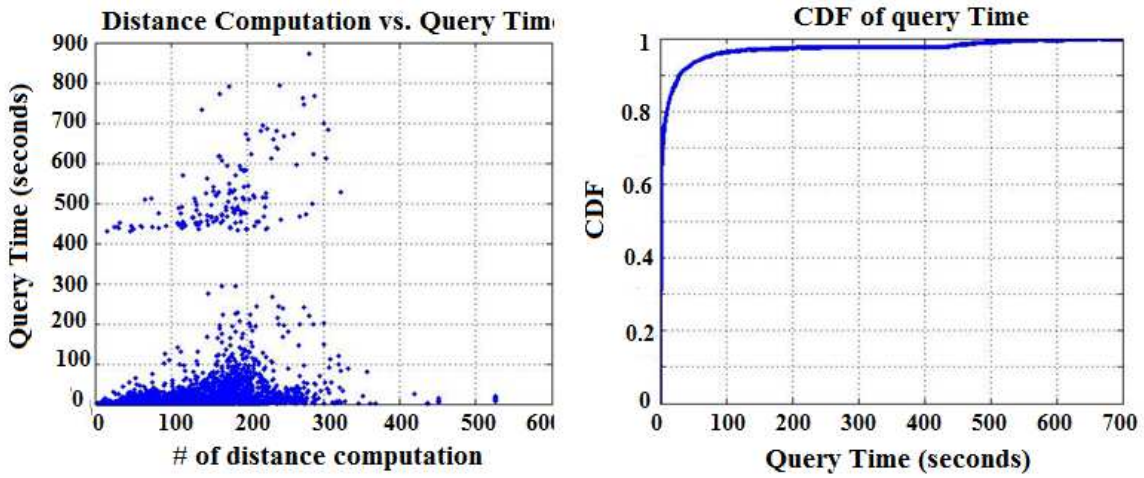


Figure 2.9: Query response time of 500 five-nearest-neighbor queries against a 100,000-malware database

2.6.5.2 Query Response Time of SMIT

Finally, we measure the response time of SMIT for K-NN queries against the entire test database, where N is set to 2 and K is set to 5. We randomly select over 500 malware files and use them to query SMIT. The response times of these queries and their cumulative distribution function are shown in Figure 2.9. The X-axis of the left figure is the number of graph-distance computations required for a query and the corresponding Y-axis is the response time in seconds for that query. From the right figure, for over 95% of all queries, the response time is less than 100 seconds, although several queries (mostly for very large malware files) incur a significantly longer delay and thus skew the overall average response time. More specifically, each 5-NN query requires, on average, 112 graph-distance computations (median is 78 and maximum is 918). The query response time ranges from 0.015 second to 872 seconds with average 21 seconds and median 0.5 second. This result demonstrates that SMIT's performance is adequate for day-to-day use even for relatively large malware databases.

2.7 Limitations and Improvements

We now discuss several limitations of the current SMIT prototype that may limit its classification effectiveness, and possible improvements to remove or alleviate them.

One way for malware authors to evade SMIT's classification is to prevent SMIT from extracting useful features by applying packers/protectors to their malware files. SMIT's classification accuracy will degrade significantly if it cannot successfully unpack packed malware files. To counter the packing problem, the current SMIT prototype employs several packer detection (PEiD, TrID) and unpack tools (SymPack), but they are by no means complete. For example, PEiD can be misled by a simple modification to a PE file's entry point. Most existing unpack tools fail to handle sophisticated packers, such as ASProtect [8], Armadillo [99] and VMProtect [108]. To improve SMIT's unpacking capabilities, we plan to incorporate generic unpackers, such as OmniUnpack [72] and Justin [41], which execute malware samples, detect the end of unpacking and then dump the process image at that instant. The extra performance overhead associated with these techniques is generally acceptable, because SMIT is mainly positioned as a back-end malware classification and analysis tool.

Second, because SMIT analyzes malware samples at the level of individual instructions and function calls, it may be susceptible to advanced obfuscation techniques. For instance, attackers may circumvent SMIT's function matching by obfuscation, such as instruction reordering, equivalent instruction substitution, import table modification (to hide the symbolic names of imported functions), etc. Alternatively, they could also modify the function-call graph by, for example, inserting useless functions into the graph, breaking existing functions into several smaller functions, inlining certain functions, etc. Although SMIT cannot completely handle all types of obfuscation, it makes these attacks more difficult. For instance, SMIT uses the edit distance between mnemonic sequences to evaluate inter-function similarity, which enables SMIT to be relatively resilient to simple code obfuscation and relocation. To defeat more sophisti-

cated obfuscation, SMIT could pre-process malware files with advanced deobfuscation techniques [85]. More importantly, because SMIT relies on *structural similarity* to match function-call graphs, changes to a few nodes in the graphs are unlikely to significantly influence the matching results. On the other hand, generating semantically equivalent but syntactically significant different malware samples is a difficult task for attackers. In the future, we plan to investigate and quantify SMIT's resilience to common obfuscation techniques.

Third, SMIT extracts function-call graphs using IDA Pro, which may occasionally fail to identify all the functions in a malware binary. IDA Pro finds function-start addresses by traversing direct function call or recognizing function prologues. As a result, if the functions are indirectly referenced or have non-standard prologues, IDA Pro may fail to identify their starting points. A more thorough approach [42] that uses a new function model based on a multi-entry control flow graph could mitigate this problem.

Finally, the dominant family metric used in SMIT may lead to false positives. Because SMIT is mainly used to help malware analysts quickly determine the maliciousness and the identity of incoming malware, it assumes that the query malware sample belongs to the same family as the majority of its nearest neighbors in the database. However, this assumption is not always valid and a false positive may occur if the distance between an input malware sample and its dominant family neighbors is too large. One way to address this problem is to apply a distance threshold in the classification process so that an input sample is classified into a malware family if and only if it is sufficiently close to the returned nearest neighbors. The optimal threshold could be chosen based on the average inter-member distance within a malware family as well as the inter-family distance between the centroids of adjacent families.

In summary, although there are ways malware writers could use to detract SMIT's overall effectiveness, SMIT is still very effective in practice against modern malware

samples, as demonstrated in Section 3.7, and thus represents a very efficient tool available for malware analysts to handle the exponentially-growing influx of malware samples as seen in recent years.

2.8 Conclusion

In recent years, attacks that target a smaller number of victims. As a result, the number of malware samples seen in the field has increased exponentially, and automating the malware processing workflow is crucial to commercial anti-virus companies such as Symantec. A critical step in malware processing workflow is to determine if an incoming sample is indeed malicious or not. A common approach taken today is to apply multiple commercial Anti-Virus scanners to a sample and convict the sample as malware if a sufficient number of Anti-Virus scanners consider it malicious. Although this approach is useful, it does not completely solve the problem, because at any point in time a significant percentage of new samples are unknown to existing Anti-Virus scanners.

This chapter describes the design, implementation and evaluation of a malware database management system called `SMIT` that implements a malware conviction approach which casts the problem of determining if a new binary sample is malicious into one of locating the sample's nearest neighbors in the malware database. `SMIT` converts each malware program into its function-call graph representation, and performs nearest neighbor search based on this graph representation.To efficiently capture the similarity among malware variants, `SMIT` supports an approximate graph-edit distance metric rather than isomorphic graph match. To efficiently support accurate and scalable nearest neighbor search, `SMIT` features a multi-resolution indexing scheme that combines a B+ tree based on high-level summary features and a vantage-point tree based on the graph-distance metric. With these techniques, `SMIT` is able to detect malware samples at a speed and accuracy level that can keep up with the current

malware sample submission rate. The main contributions of this work include:

- an efficient graph-distance computation algorithm whose result closely approximates the ideal graph-edit distance metric;
- a multi-resolution indexing scheme that supports efficient pruning through a combination of exact indexing based on summary features and nearest-neighbor indexing based on graph-edit distance; and
- A fully working `SMIT` prototype and a comprehensive performance study of this prototype that demonstrates its efficacy and scalability with a 100,000-malware database.

CHAPTER III

MutantX: Scalable Malware Clustering Based on Static Features

3.1 Introduction

Over the last few years, we have witnessed a significant increase of malware threats. According to the Symantec's latest Internet Threat Report, the number of new malicious code signatures created in 2009 has reached 2,895,802 which is a 71% increase over the 2008 number and accounts for 51% of all the malicious code signatures. This exponential growth of malware samples has already outpaced the current manual analysis techniques, causing anti-virus (AV) companies to face a major challenge, "how to process this huge influx of incoming samples and accurately label them?" Typically, AV companies receive several thousands of suspicious samples everyday. It is practically impossible to manually analyze—as AV companies do—such a huge number of samples, thus leaving a large percentage of samples unlabeled. This slow rate of processing incoming malware samples delays the signature distribution and ultimately results in poor detection of malware in the wild. One possible solution to this problem is to *automatically* cluster malware samples and assign them labels according to their similarities. The intuition behind this is that malware programs bearing significant similarities are likely to have been derived from the same code

base, and hence from the same malware family. One can group similar samples into a cluster and label it with high accuracy by analyzing only a few representative samples from the cluster. Moreover, the label of a new sample can be automatically derived if it is determined to belong to an existing known cluster. In this chapter, we design, implement and evaluate `MutantX`, a novel and scalable system, that can efficiently cluster a large number of malware samples into families based on their static features, i.e., code instruction sequences.

Most existing malware clustering/classification systems are based on dynamic behavioral features. These dynamic-analysis systems operate by running malware samples in virtual or sandboxed environments, monitoring their execution and extracting their run-time behaviors in terms of API or system call traces [10, 13, 88]. The major benefit of using dynamic behavioral features is that they are less susceptible to mutation schemes, such as run-time packers or binary obfuscation, frequently employed by malware writers to avoid static analysis. Albeit very useful in practice, approaches based on dynamic behavioral features suffer from several limitations as follows. First, they may have only limited coverage of an application's behavior, failing to reveal the entire capabilities of a given malware program. This is because for a particular execution run, a dynamic analysis can only capture API or system call traces corresponding to the code path that was taken during that particular execution. Different code paths may be taken in different runs, depending on the program's internal logics and/or external environments. More commonly, many malware include triggers in their programs and exhibit an interesting behavior only when certain conditions are met. Typical examples include bot programs that wait for commands from their botmasters, and malware programs designed to launch attacks at or before a certain date and time. These trigger-based malware generate very few repeatable run-time traces unless specific conditions are met. Second, dynamic-analysis is inherently resource-intensive and does not scale well, limiting their coverage. To process the

sheer number of malware samples collected everyday with the limited computational resource, a dynamic-analysis system can only execute and monitor each sample for a short period of time, e.g., a couple of minutes. Unfortunately, this time is often too short for typical malware programs to reveal all their behaviors.

In this chapter, we present `MutantX`, a new and practical system that exploits static features of code instruction sequences for efficient and automatic malware clustering and labeling. `MutantX` is motivated by the common observation that a large portion of today's malicious programs are file-level variations of a small number of malware families and tend to share similar instruction sequences in their binaries. Analysis of static features of malware offers several unique benefits. For example, it has the potential to cover all possible code paths of a (malicious) program, yielding more accurate clusters based on the entire functionalities of the programs. Moreover, approaches based on static features are much more scalable than their dynamic counterparts, as they do not require resource-intensive execution and time-consuming monitoring of malicious programs. This is particularly important for AV companies to process a rapidly-increasing number of new malware samples. Unfortunately, the static-feature-based approaches are not without limitations of their own. It is well-known that they suffer from run-time packing and obfuscation. Therefore, the goal of `MutantX` is not to replace existing dynamic-behavior-based systems, but to complement and collaborate with them to achieve higher clustering accuracy and better coverage of malware programs.

`MutantX` features a unique combination of techniques to address the deficiencies of static-feature-based malware clustering. First, it employs an efficient encoding mechanism that exploits the IA32 instruction format to encode malware binaries into a opcode sequence, facilitating the extraction of N-gram features. Second, it applies a hashing-trick on the extracted N-gram features that help the clustering algorithm handle very high dimensional features. Finally, it tailors a generic unpacking tech-

nique to handle commonly-seen run-time packers so that the clustering algorithms may be applied to a larger set of malware samples. We have successfully implemented a fully-automated prototype of `MutantX` and evaluated its performance using a database of more than 100,000 distinct malicious programs. Our evaluation results demonstrate that `MutantX` can effectively create clusters corresponding to malware families which can help improve the accuracy of malware labeling and reduce/remove the manual analysis effort, thereby enabling faster response to new malware threats.

3.2 Related Work

Malware poses one of the severest threats to computer systems and the Internet. As a result, automatic malware clustering and classification have recently attracted considerable attention from the security industry and research community. Various schemes have been proposed to tackle this problem based on the dynamic behavior and static features of malware.

The major benefit of dynamic-analysis approaches is the ease of handling packed and obfuscated malware samples. Dynamic analysis works by executing malware programs in a virtual or sandboxed environment and collecting their behavior in terms of system or API calls and their arguments. Lee and Mody [64] proposed use of a sequence of events (e.g., registry and file system modifications) to capture rich behavioral semantics. They applied a nearest-neighbor approach and assigned the same class label to a new malware code as that of its nearest neighbor in a set of known samples. Rieck *et al.* [86] embedded the run-time behavior, such as copy file and create processes, into feature vectors according to each feature's frequency and applied SVM (Support Vector Machine) to learn and classify unknown samples. One limitation of this approach, as noted in [13], is that it uses supervised learning techniques and thus requires labeled training sets. Later, Bailey *et al.* [10] defined the malware behavior as non-transient state changes caused to the system and applied a hierarchi-

cal clustering algorithm to group similarly-behaving malware samples. Unfortunately, the complexity of this clustering algorithm is $O(n^2)$, limiting its applicability only to a small number of samples. To address this problem, Bayer *et al.* [13] and Rieck *et al.* [88] developed different methods to make the clustering algorithm scalable. Bayer *et al.* [13] applies locality-sensitive hashing (LSH) to efficiently compute an approximate hierarchical clustering with a significantly smaller number of distance computations. By contrast, Rieck *et al.* [88] developed a prototype-based clustering algorithm that reduces the runtime complexity by performing clustering only on representative samples (i.e., prototypes). Comparing with the approximate LSH clustering, a prototype-based algorithm facilitates the analysis of behavior groups because each prototype corresponds to regular malware samples [88]. In `MutantX`, we adopt the same prototye-based algorithm as in [88] because of its efficiency and explicit expression of malware features.

Static analysis, on the other hand, extracts various features from malware binaries and use them as the basis for analysis, classification and detection. For example, Christodorescu *et al.* [22] extracted unique malicious patterns from disassembled malware that are resilient to obfuscation. Wicherski [113] utilizes static features extracted from PE headers, e.g., raw size, entry point, import table and section characteristics to group malware. Such PE header features have also been used by Perdisci *et al.* [81] to accurately differentiate between packed and unpacked malware samples. Karim *et al.* [53] took a different approach and studied the malware evolution by creating phylogeny models of malware families based on N-gram and N-perm on assembly instructions. Similar features have also been used in [56] to perform supervised learning with various methods, such as naive Bayes, decision trees, SVM, etc., and validate their effectiveness in classifying samples. `MutantX` falls into the static-analysis category because it relies on malware features extracted from the malware code instructions to cluster samples. The main difference of `MutantX` from previous

approaches is its unique combination of techniques that ensures the scalability to large malware datasets corresponding to the huge number of current malware files. Although malware sets of a similar size have been studied with dynamic-behavior-based clustering [13], static analysis is still necessary and sometimes advantageous because it does not suffer from the limited coverage of dynamic analysis. Another system similar to `MutantX` is BitShred independently developed by Jang *et al.* [50] and also focuses on improving the scalability of malware comparison and triage on a large scale. BitShred developed a fast code comparison algorithm based on hashes of byte sequences in code section and made use of distributed computing resources, i.e., hadoop and MapReduce to achieve a high throughput in binary comparison and good scalability.

3.3 Architecture

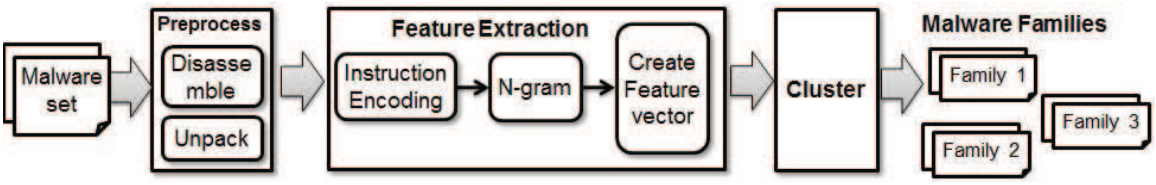


Figure 3.1: `MutantX` system overview

This section presents an overview of `MutantX` as shown in Figure 3.1. At a high level, `MutantX` takes a set of malicious or suspicious program samples as input and extract their features using static analysis to avoid the computation overhead and maximize code coverage. Specifically, `MutantX` first uses existing tools (e.g., PeID, TrID, SymPack) to identify malware files that have been processed by the binary packing tools such as UPX [106], ASPack[7], and other customized packers. These files will be unpacked with a generic unpacking technique tailored for `MutantX`. Together with samples that are in their original binary (not packed), unpacked binaries

are disassembled to extract their code instructions. These pre-processing steps ensure that `MutantX` can successfully extract the features inherent to malware families without influence of encryption or compression. After their pre-processing, all malware samples are passed to the second component of `MutantX` and processed with three algorithms to extract their representative features: (1) *Instruction Encoding* for converting each instruction to a sequence of encoded operation codes that capture the underlying semantics of the programs, (2) *N-gram analysis* for constructing feature vectors that allow computation of similarities between any programs, and (3) *Hashing Trick* for compressing the feature vectors, significantly improving the speed of similarity computation while incurring only a small penalty in clustering accuracy. Finally, a prototype-based clustering algorithm is applied on the set of compressed feature vectors and partitions samples into different clusters, each representing a group of similar malicious programs.

3.4 Generic Unpacking Algorithm

For its simplicity and effectiveness, run-time packing is most commonly used by malware programs to circumvent anti-virus detection tools and evade their static analysis. More than 50% of all malware programs are estimated to be packed by some type of packers. A typical packer like UPX works as follows. When compressing a PE binary (the executable file format used by the Windows operating systems), UPX merges all of its (both codes and data) sections as well as the original PE header, compresses them into a single section, and creates a new PE binary containing the compressed data followed by the unpacker code. When athe packed program is executed, it first invokes the decryption routine to restore the original program codes into memory and then jump to the first instruction of the unpacked codes (i.e., the original entry point) to resume execution. As a result, packing enables malware programs to disguise their malicious instructions in random-looking data while keeping

the original functionality intact. Since MutantX relies on similarity between original code instructions to cluster malware samples, As a result, it is imperative for MutantX to handle packing correctly and efficiently.

While there exist static unpacking tools such as SymPack and ArmaGedoon, they are often targeted specifically at one or a few packers, and are not optimal. The practical use of these unpacking tools is limited for two reasons. First, it entails significant investments in the engineering effort for each distinct packer, because human experts have to manually reverse-engineer packers and develop unpacking tools for each and every of them. As more packers appear in the wild, the cost of continually updating unpacking tools is expected to grow over time. Second, requiring manual analyses will incur a (often significant) delay between the appearance of a new packer and the creation of unpacking tools for the packed malware, leaving a dangerous time window for malware to evade the detection.

MutantX thus adopts a generic unpacking mechanism and tailoring it to meet the particular need for malware clustering. Its basic idea is to exploit the inherent properties of an unpacking procedure, i.e., a packed binary has to output the unpacked code into some memory space and transfer control to the modified memory locations to continue execution. By continual monitoring of memory access, we can learn that some form of unpacking, self-modification or on-the-fly code generation occurs when the program executes instructions in a memory address that has been written after the program was loaded. These written-then-executed memory locations are likely to contain the original (unpacked) program codes and thus are targets to be analyzed by MutantX.

The unpacking component of MutantX is derived from Justin [41] and exploits the physical non-execution (NX) support in modern x86 CPUs and Windows OS to track memory page status. It consists of a kernel driver responsible for tracking system calls and a user-level component that is injected as a remote thread into a program's

address space (Figure 3.2). The unpacking component of `MutantX` achieves two goals: (1) dump the memory image at an appropriate time when the binary is likely to finish unpacking, and (2) determine the correct original entry point (OEP) of the unpacked program. Finding the correct OEP is critical for program disassembly and feature extraction, because a wrong entry point may cause MutnatX to miss all the code instructions between the original and the wrong entry points (if there is no other reference to this portion of code). In addition, because the disassembling starts from the entry point, if the entry point is incorrectly set in the middle of an instruction, the disassembler will either fail or generate completely wrong assembly codes.

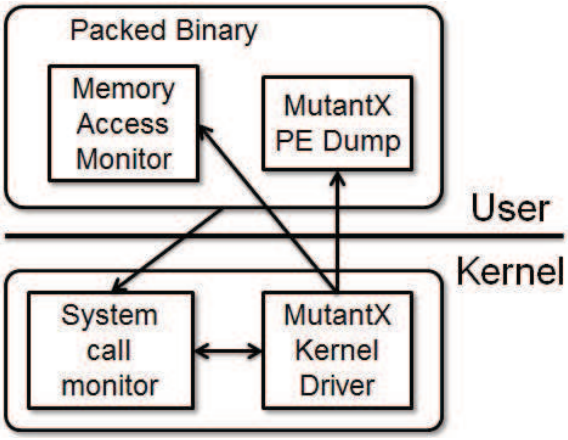


Figure 3.2: `MutantX`'s generic unpacking component

`MutantX` unpacks a packed binary as follows.

1. It loads the packed program into memory, suspends its execution and communicates with the `MutantX` kernel driver to inject the user-level hooking DLL (Dynamic Linked Library) into the process' memory space.

2. Inside the hooking DLL, `MutantX` registers an exception handler with the OS to intercept and process memory access exceptions generated by the unpacker violating the $W \oplus X$ *write-xor-execute* policy.

Algorithm 2 `MutantX` generic unpacking algorithm

Input: A packed binary program B
Output: a reconstructed PE file containing unpacked program codes
STEP 1:
Load the packed program into memory
for all p in the program's memory pages **do**
    $Permission(p)| = \tilde{W}$
end for

STEP 2:
while B is running **and** $T_{runtime} < T_{thresh}$ **do**
    a: The address of the page fault
    t: The page fault type $t \in$ {WRITE, EXECUTE}
    $p \leftarrow Page(a)$
    if t = WRITE **then**
        $Permission(p)| = (W|\tilde{X})$
        $LAST\_WRITTEN(p) \leftarrow$ current time
    end if
    if t = EXECUTE **then**
        $Permission(p)| = (\tilde{W}|X)$
        $LAST\_EXEC(p) \leftarrow$ current time
        $ADDR\_EXEC(k) \leftarrow a$
    end if
end while

STEP 3:
Dump process memory
reconstruct B' by setting OEP to be $ADDR\_EXEC(k)$
$k = \arg\min_k(LAST\_EXEC(k) > \max(LAST\_WRITTEN(i))$
return B'

3. It marks all the memory pages of the loaded program as executable but non-writable, and resumes its execution.

4. During the execution, when the unpacker attempts to restore unpacked codes into memory, a write exception will occur on a non-writable page. `MutantX` marks the page as dirty and changes its permission to writable but non-executable.

5. When the unpacker accesses the the newly-generated code for execution, the absence of execution permission causes an access exception. `MutantX` intercepts such exceptions and records addresses where the exceptions had occurred. For a simple unpacker that completes unpacking the entire program before transferring control to it, this memory address corresponds to the original entry point. However, this is not necessarily true for more sophisticated packers (e.g., self-modifying code that may rewrite to the same memory location). Therefore, `MutantX` removes the write permission from these memory pages again, grants execution privilege and continues execution.

6. To ensure the integrity of the $W \oplus X$ policy, `MutantX` tracks all system calls that change the permission of a memory page such that when the program attempts to modify attributes of the pages in ways that conflict with `MutantX` settings (e.g., granting write permission on non-writable pages), `MutantX` pretends that the operation is successful while keeping the original settings. `MutantX` also monitors dynamic allocation of memory pages and removes their write permission to track unpacking on these pages.

7. `MutantX` dumps the process memory image either at the end of program execution (by hooking LdrProcessShutdown function in NtDll.dll) or after a certain period of time. The basic intuition behind this is that after the program has been running for a sufficient amount of time (e.g., 30 seconds to 1 minute), it is

fairly safe to assume that the program has finished unpacking and restores the original codes in memory.

From the dumped memory image, `MutantX` creates a new PE header and reconstructs a valid PE file so that a standard disassembler can extract instructions. The major challenge is to identify the correct entry point of the original program in order to ensure proper disassembly. `MutantX` exploits the same $W \oplus X$ policy to address this problem. For a simple packer like UPX, which starts the execution of the original program after restoring the entire program in the memory, the OEP of the original program is simply the address of the dirty memory page (i.e., memory page that has been modified and marked as non-executable) where the first execution exception occurs. Unfortunately, as adversaries are becoming increasingly aware of the generic unpacking techniques, they have developed various evasion schemes that complicate the detection of OEP. A typical method is to fake end-of-unpacking by writing some instructions into a reserved memory page, transfer control to it, and jump back to the unpacker code. This creates an illusion to unpacking tools that unpacking has ended, and misleads the unpacking tools to conclude with the wrong address of the entry point. More sophisticated unpackers adopt incremental unpacking where the unpacker decrypts a few payload instructions and executes them, then decrypts some more and executes them, and so on. In such a scenario, detecting the first execution exception is not enough because only part of the original program has been unpacked. `MutantX` develops the *Last Modification First Execution* (LMFE) heuristic as detailed below.

Our basic idea is to keep track of time when the last write exception and a subsequent execution exception occur on each memory page, so `MutantX` can identify the unpacker's attempts to overwrite to the same memory page multiple times, in which case the previous modification and execution on the page are likely to be spurious, trying to fool the unpacking tools. More specifically, for each memory page,

`MutantX` keeps a record of time when it was last modified (i.e., a write exception occurred on the page), denoted as $LAST\_WRITTEN$; the time when the last execute exception occurred, denoted as $LAST\_EXEC$; and the address $ADDR\_EXEC$ where the exception had occurred. At any point of program execution, there are three types of memory page as follows.

Type I: memory pages that have valid $LAST\_WRITTEN$ and $LAST\_EXEC$, i.e., pages that are both modified and executed.

Type II: memory pages that have valid $LAST\_WRITTEN$ but not $LAST\_EXEC$, i.e., pages that are modified but not executed. They could either be data or code section pages that have not yet been executed.

Type III: memory pages that have neither valid $LAST\_WRITTEN$ nor valid $LAST\_EXEC$. These could be initialized data-section pages or unpacker-code pages.

Essentially, type-I memory pages are those that hold the unpacked instructions and thus contain the original entry point of the unpacked program. When dumping the process memory, `MutantX` uses the following algorithm to pinpoint the correct OEP. Let $P(i),\ i = 1..n$ represent all type-I memory pages of the packed program and $LAST\_WRITTEN(i)$, $LAST\_EXEC(i)$ and $ADDR\_EXEC(i)$ represent the timestamps of the last write exception, last execution exception and address where the exception occurs for page $P(i)$, respectively. Then, the original entry point is $ADDR\_EXEC(k)$ in the memory page $P(k)$ where

$$k = \arg\min_{k}(LAST\_EXEC(k) > \max(LAST\_WRITTEN(i)) \tag{3.1}$$

where $i = 1..n$. In other words, $P(k)$ is the first memory page that is executed after all type-I memory pages have been written. Below we show that `MutantX` is able to find the OEP even when the packers try to fool the `MutantX` using spurious write-and-execute sequences or mutli-layer packing.

Proposition 1. For k satisfying Eq. (3.1), $ADDR\_EXEC(k)$ is the correct OEP of the

original program if the program is packed with simple packers or packers that write random instructions into memory, execute them and jump back to the unpacker code to complete the real unpacking.

Proof. Let us first look at a simple packer like UPX that restores the entire program into memory before transferring control to it. Let $P(j)$, $j = 1..m$ denote memory pages where the unpacker writes to the original program. Without loss of generality, we can assume that the contents are written sequentially from $P(1)$ to $P(k)$, meaning that $LAST\_WRITTEN(1) < LAST\_WRITTEN(2) < \ldots < LAST\_WRITTEN(m)$. When the packer starts executing the restored program by jumping to the OEP after it finishes unpacking, execution exceptions will first occur on the memory page $P(k)$ that contains the OEP, i.e., $LAST\_EXEC(k) > LAST\_WRITTEN(m)$ and $LAST\_EXEC(k) < LAST\_EXEC(j) \forall j \neq k$. As a result, $ADDR\_EXEC(k)$ is the correct OEP.

Second, assume a packer has the following spurious unpacking sequence that writes arbitrary instructions into some memory page, executes them and, at the end of execution, returns to the unpacker code. Such a routine may be called multiple times during the whole unpacking process. As a result, an unpacking tool will fail if it assumes the end-of-unpacking at the first (or first few) execution exception. `MutantX` is resilient to this type of evasion by enforcing the invariant that the execution exception on the OEP must succeed all the write exceptions. For example, when the spurious unpacking routing touches memory page $P(s)$, `MutantX` records $LAST\_EXEC(s)$ and marks $P(s)$ as executable but non-writable. Then, the unpacker resumes the normal unpacking and writes the original program to memory page $P(t)$ (t could be any memory page including s). This creates a new write exception on $P(t)$ at timestamp $LAST\_WRITTEN(t)$. Note that because $LAST\_EXEC(s) < LAST\_WRITTEN(t)$, `MutantX`determines s to not contain the OEP. In contrast, after the packer finishes unpacking and transfers control to the real

OEP, the execution exception satisfies Eq. (3.1). By keeping $ADDR\_EXEC$ up-to-date and pointing to a valid instruction, `MutantX` is able to recognize the real OEP accurately. Same arguments hold for multi-layer packing because the write exceptions of code pages will always precede the executable exceptions caused by jumping to the OEP. □

After pinpointing the real OEP, `MutantX` reconstructs the PE structure of the memory images by setting up a proper PE header, ensuring a correct starting point for disassemblers. Algorithm 2 summarizes the `MutantX`'s generic unpacking algorithm.

3.5 Feature Extraction

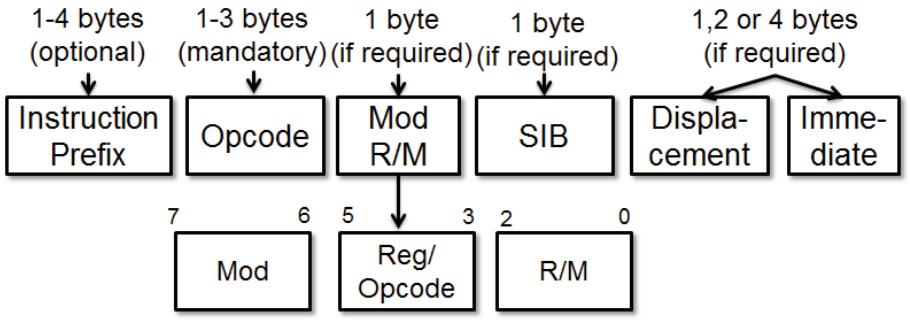


Figure 3.3: x86 instruction format

Given unpacked malware binaries, `MutantX` uses a custom-developed disassembler to break them down into a sequence of machine instructions that are used for feature extraction. The key step taken by `MutantX` is the encoding of machine code instructions suitable for the comparison of similarity between malware samples. The challenge in computing similarities is how to handle variations in the syntax of instructions. Malware often undergo changes for various reasons, such as mutation, polymorphism, and obfuscation. As a result, ensuring exactness in comparing instructions will not tolerate any variation in the syntax. At the other extreme, correctness

is compromised if all forms of variation are tolerated. `MutantX` strikes a balance between these two extremes by exploiting the x86 instruction format (Fig. 3.3) and uses the opcode as a succinct representation of the instruction semantics.

Opcode Length	Opcode	Binary	Assembly Code
1 byte	83	83 F0 4F	XOR EAX, 4Fh
	83	83 C0 4F	ADD EAX, 4Fh
	BB	B8 4F 00 00 00	MOV ECX, 4Fh
2 bytes	0F 22	0F 22 C3	MOV CR0, ECX
	0F 00	0F 00 C0	SLDT AX
	0F 00	0F 00 D0	LLDT AX
3 bytes	0F 38 1D	0F 38 1D D8	pabsw mm3, mm3

Table 3.1: Opcodes of varying lengths

Using opcodes—instead of other features such as API call sequences, control flow graphs or binary sequences—offers several benefits. First, opcode generalizes well to represent variants of a malware family. Malware samples in the same family are observed to have been derived from the same code base and thus share similarities in their instructions. However, due to relinking, rebinding and rebasing, the operands of instructions tend to change among the variants. As a result, using opcodes and ignoring the operands (i) make `MutantX` more resilient to low-level mutations while providing a meaningful characterization of semantics and (ii) reflect the functionality of the malware programs. Second, compared to the previous approaches that use mnemonic sequences (e.g., mov, push), the opcode sequence is more accurate in representing features. Mnemonics sometimes *overly* generalize the underlying CPU operations, causing many different instructions (or instructions with distinct semantics) to appear similar. To illustrate this, consider all the instructions in Table 3.2. Although all of them have the same mnemonic (i.e., mov), the underlying semantic is drastically different. For instance, moving a value to a control register often indicates a critical OS operation, such as interrupt control, switching addressing mode, paging control, etc., which intuitively should not be treated same as moving a value

between one register and another. Ideally, moving from memory to a register (memory load operation) should be considered as a distinct operation from that of moving from a register to memory (memory store operation). Unfortunately, using mnemonics would cause all these distinct instructions to be represented with a single feature (i.e., mov), which may lead to an accidental similarity between feature re-orientations and then to false positives. On the other hand, features based on opcode as used in MutnatX provide a distinct representation for each of these instructions, providing better distinguishability and hence better accuracy in malware clustering.

Opcode	Instruction	Description
89	MOV r/m32, r32	Move from reg to mem/reg
8B	MOV r32, r/m32	Move from mem/reg to reg
B8	MOV r32, imm32	Move immediate val to reg
0F 20	MOV r32, CR0-CR4	Move from control reg to reg
0F 22	MOV CR0-CR4,r32	Move from reg to control reg
0F 21	MOV r32, DR0-DR7	Move from debug reg to reg
0F 23	MOV DR0-DR7,r32	Move from reg to debug reg

Table 3.2: Opcodes provide fine-grained representations of instruction semantics (reg: register, mem: memory)

Unfortunately, using opcodes to represent features is not as easy as they may appear. Since x86 architecture is a complex instruction set computer (CISC), the opcodes on x86 can be 1 byte, 2 bytes (prefixed by 0x0F) or 3 bytes (prefixed by 0x0F38 or 0x0F3A) of length as shown in Table 3.1. Moreover, some 1- and 2-byte opcodes belong to an "extension group" or "opcode group" where 3rd, 4th and 5th bits of ModR/M work like a 3-bit "sub opcode" (Fig. 3.3). For example, "83 F0 4F" is disassembled to "XOR EAX, 4Fh" while "83 C0 4F" is "ADD EAX, 4Fh", even though their primary opcodes are both 83 (note that their sub opcode are 110 and 000, respectively). To handle these irregular formats of instruction opcodes and facilitate feature extraction, `MutantX` encodes each x86 opcode into a standardized intermediate format (IF) where an encoded opcode is 2 bytes long. The encoding method is summarized in Algorithm 3. Its basic idea is to manipulate the 1-byte

prefix in front of the primary opcode byte in the way that guarantees no collision between encoded opcodes, because different types of opcode in the x86 instruction set occupy disjoint encoding spaces in the standardized opcode set. Specifically, the encoding of 1-, 2-, and 3-byte opcodes ranges from 0x0000 to 0x07FF, 0x0F00 to 0x16FF, and 0x2500 to 0x26FF, respectively. The encoding scheme can also be easily extended to accommodate any new machine language instruction that may be added later.

With this encoding scheme, a program can be represented as a sequence of encoded opcodes (Fig. 3.4). We then use the standard N-gram analysis to characterize the content of a malware program, i.e., moving a fixed-length window over the sequence and consider a subsequence of length N at each position. The resulting N-gram of opcodes reflects short instruction patterns and thus implicitly captures the underlying program semantics. To construct a feature vector from the opcode N-grams, we consider the set S of all possible N-grams, defined as $S = \{(o_1, o2, \ldots, o_N)|o_i \in \mathcal{O}, 1 \leq i \leq N\}$ where $\mathcal{O}$ is the set of all encoded opcodes. Then, a program can be converted to a feature vector V in an $|S|$-dimensional vector space ($|S| = |\mathcal{O}|^N$) where each dimension number of *occurrence* of one particular opcode N-gram. This way, the similarity between two programs can be calculated geometrically in the vector space, which ultimately enables efficient analysis and clustering of malware samples. In `MutantX`, the similarity between malware samples is characterized using the Euclidean distance between feature vectors in the vector space. That is, given two malware samples m and n, the distance between them is defined as:

$$d(m, n) = \|V_m - V_n\| = \sqrt{\sum_{i=1} |S|(V_m(i) - V_n(i))^2}$$

. Compared to the other approaches that compute the similarity (e.g., locality-based hashing), geometric assessment of similarity between malware in the vector space

Algorithm 3 X86 opcode encoding method

Input: x86 instruction I
Output: 2-byte encoded opcode $(C1, C2)$

// list1: all 1-byte opcodes that belong to an extension group
list1 = [0x80, 0x81, 0x82, 0x83, 0xC0, 0xC1, 0xD0, 0xD1, 0xD2, 0xD3, 0xF6, 0xF7, 0xFE, 0xFF]

// list2: all 2-byte opcodes that belong to an extension group
list2 = [0x0F00, 0x0F01, 0x0FC7, 0x0F71, 0x0F72, 0x0F73]

$P_o \leftarrow$ opcode of I
$Sub_o \leftarrow$ sub opcode of I *// 3rd, 4th and 5th bits of ModR/M*

if length$(P_o) = 1$ **then**
    // 1-byte opcode
    if $P_o \in$ list1 **then**
        $C1 \leftarrow Sub_o$; $C2 \leftarrow P_o$
    else
        $C1 \leftarrow 0x00$; $C2 \leftarrow P_o$
    end if
end if

if length$(P_o) = 2$ **then**
    // 2-byte opcode
    if $P_o \in$ list2 **then**
        $C1 \leftarrow 0x0F + Sub_o$; $C2 \leftarrow P_o$
    else
        $C1 \leftarrow 0x0F$; $C2 \leftarrow P_o$
    end if
end if

if length$(P_o) = 3$ **then**
    // 3-byte opcode
    if $P_o[2] = 38$ **then**
        $C1 \leftarrow 0x25$; $C2 \leftarrow P_o$
    else
        $C1 \leftarrow 0x26$; $C2 \leftarrow P_o$
    end if
end if
return $(C1, C2)$

provides the benefit of *explicit feature representation* [88] where the importance or contribution of each N-gram in clustering similar malware can be traced back to its original code patterns. For N-grams that potentially correspond to inherent characteristics of a malware family, e.g., those that frequently appear within a family but rarely occur in others, their original code segments can be traced back and used signatures to detect malware variants.

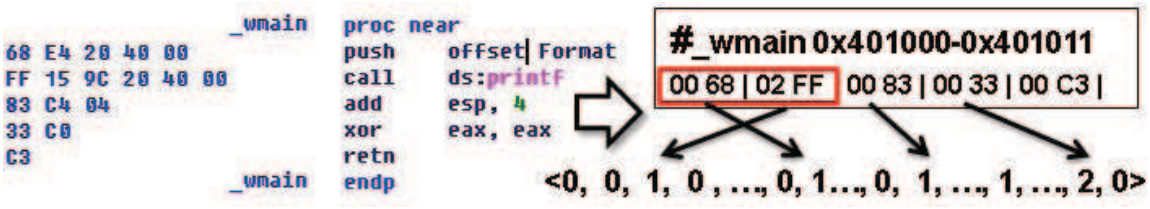


Figure 3.4: Encoding a function into a standardized format

3.6 Clustering Algorithm

A malware cluster is a collection of malware samples that share some common traits but differ from those of other clusters. Considering the enormous number of malware in the wild and their exponentially growing speed, a reasonable goal of `MutantX` is to process hundreds of thousands or even a million malware files sufficiently fast. Unfortunately, classic clustering algorithms such as hierarchical and partitioning-based clustering, e.g., K-Means or K-Medoids—although they have been successfully applied to cluster malware behavior [10] and create phylogeny trees [53]—incur a the time complexity at least quadratic in the number of samples that in practice does not scale to the `MutantX`'s target. For efficiency, `MutantX` utilizes (1) a hash kernel that compresses the high dimensional feature vector into a low dimensional space, and (2) a prototype-based clustering algorithm that has close-to-linear runtime complexity.

3.6.1 Hashing Kernel

Kernel methods [92] are powerful tools used in machine learning and pattern recognition to allow operation in the high-dimensional feature space without ever having to compute the coordinates of the data in the space. This is particularly useful when the input data has a non-linear decision boundary but can be linearly separated in a high dimensional feature space. In general, given input data $x_1, \ldots, x_n \in \mathcal{X}$ for some input domain $\mathcal{X}$, the kernel methods compare two input data as:

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

where ϕ is the mapping function from $\mathcal{X}$ to some feature space.

In `MutantX`, however, we have encountered the opposite problem: the original space has a very high dimension and the training and testing data are prohibitively large in size and in dimension. For this scenario, several researchers have recently proposed a *hashing-trick* which *hashes* the high dimensional input vector $x \in \mathbb{R}^n$ into a lower dimensional feature space $\mathbb{R}^m$ with mapping function $\phi : \mathcal{X} \rightarrow \mathbb{R}^m$. Since $m \ll n$, the hashing trick reduces a feature vector to a more compact representation, allowing the clustering algorithm to handle a large volume of data and save both computational time and memory requirements. Previous research has shown that the hash kernel approximately preserves the inner product and is generally applicable [55], because the penalty incurred from using a hash for reducing dimensionality only grows *logarithmically* with the number of objects and classes[93].

To use the hashing kernel, instead of assigning each N-gram a unique index, `MutantX` applies a *uniform* hash function $h : \{\text{N-gram}\} \rightarrow [1..m]$ that hashes N-gram directly into a position in the feature vector of length m. In case of a collision where two or more N-grams map to the same position, the sum of their counts is used as the value in the new vector. More formally, for malware M and M', let v and v'

represent their original feature vector extracted from the encoded opcode sequences and ξ denote the mapping from the N-gram $(o_1, o2, \ldots, o_N) \in S$ to the index in v. We define the hash feature map ϕ as

$$\phi_i(v) = \sum_{l:h(l)=i, l \in S} v(\xi(l))$$

and the distance between M and M' as

$$d_\phi(M, M') = \|v - v'\|_\phi = \|\phi(v), \phi(v')\|.$$

The length of the new (low dimensional) feature vector can be selected according to the size of the available memory space. Choosing a smaller m leads to a smaller memory footprint and fast vector comparison. However, decreasing m reduces the number of bins in which the hash function can place the different N-gram and consequently increases the collision possibility. In other words, over-compression of feature vectors will negatively impact the clustering accuracy.

3.6.2 Prototype-Based Clustering

The main bottleneck in clustering is comparison ofg malware samples. Classic clustering algorithms are typically super-linear in the size of the input data and thus do not scale well for a large number of samples. For example, the worst-case running time for two most widely-used clustering algorithms k-means and agglomerative hierarchical clustering are $O(n^{kd})$ [6] and $O(n^2 \log n)$ [71], resulting in the computation time that is prohibitively large for the number of malware samples we have to deal with. To address this scalability problem, `MutantX` adopts the prototype-based linear-time clustering algorithm originally designed in [88] for clustering dynamic behavior of malware programs.

Prototype-based algorithms belong to the type of unstructured and model-free

methods for clustering and pattern matching. Despite their simplicity, they are empirically shown to be very effective and often one of the best performers in real data [43]. Prototype-based clustering utilizes a set of prototypes—data points that are typical for a group of homogeneous data samples—to organize input data. Each prototype is assigned a class label and other data points are associated with their closest prototype in the feature space. By performing most computation on a relatively small set of prototypes, the algorithm avoids expensive pair-wise comparisons between original data points, and thus improves the clustering speed significantly. Specifically, the algorithm used in [88] consists of two steps: prototype extraction and clustering using prototypes.

Prototype extraction. Since clustering algorithms essentially rely on a set of prototypes to organize the original input data, the effectiveness of clustering hinges on the choice of the prototypes. Well-positioned prototypes can accurately capture the distribution of input data types and create accurate class boundaries in the feature space. Unfortunately, determining the optimal number and positions of prototypes has been shown to be NP-hard. For scalability consideration, in [88], an approximate algorithm by Gonzàlez [39] was used to iteratively select prototypes from the input data points $v_i,\ i = 1..n$. The algorithm maintains a data structure $d[i]$ that stores the distance from all the data points v_i to their nearest prototype. At each iteration, the data point $v_k, 1 \leq k \leq n$ that has the biggest $d[k]$, i.e. the shortest distance to its closest prototype, is selected as the next prototype. With this new prototype, the algorithm updates $d[j]$ for all v_j whose distance to the new prototype v_k is smaller than the original $d[j]$ ($d[j] = \|v_j, v_k\|$ if $d[j] > \|v_j, v_k\| \forall j \in [1..n]$). This selection process is repeated until the distance from all the data points to their nearest prototype is smaller than a predefined threshold P_{max}, i.e., $max(d[j]) \leq P_{max}$. In other words, all the data points are located within a certain radius from their closest prototypes. The run-time complexity of this algorithm is $O(kn)$ where k is the number of prototypes

selected. Since k only depends on the distribution of the data (in this case, k is proportional to the number of similar malware groups or families), with a reasonable choice of P_{max} the algorithm is linear in the number of input data n.

Clustering with prototypes. Instead of working on the huge number of original data points, the algorithm performs standard agglomerative hierarchical clustering on the set of prototypes selected in the previous step. As each prototype can be viewed as a reasonably good representation of data points in its close proximity (within a radius of P_{max}), the algorithm avoids expensive pairwise distance computation between the original data points without too much loss in the overall accuracy. Specifically, the algorithm starts with individual prototypes as singleton clusters, successively merges two closest clusters, and terminates when the distance between the closest clusters is larger than a predefined distance threshold Min_d. Then, prototypes within the same cluster are assigned the same label and subsequently propagate the label to their associated data points. The run-time complexity of hierarchial clustering step and propagation step are $O(k^2 log\ k)$ and $O(n)$, respectively. Compared to the $O(n^2 log\ n)$ complexity of applying an exact hierarchical clustering algorithm on the original data points, this algorithm achieves a significant speed-up, with a factor of at least $(n/k)^2$.

3.7 Experimental Evaluation

In this section, we evaluate the efficiency and accuracy of `MutantX` in clustering malware samples. We have conducted experiments on two data sets: (1) a reference data set containing 4821 malware files whose family labels are generated by security experts and thus more reliable; and (2) a large malware data set collected from an online malware archive [110] which comprises 132,234 malware samples with potentially unreliable labels derived from AV scanners. The reference data set is collected and analyzed by Symantec malware analysts in April 2009. It includes malware samples from 20 different families and their detailed distribution is given in Table 3.3.

Considering its reliable labeling, the reference set is used to evaluate and fine-tune the empirical parameters for the MutantX's clustering engine while the large malware set is used to assess the scalability of MutantX.

Family	#	Family	#	Family	#
Pilleuz	500	Bredolab	301	Tidserv	59
Koobface	496	Vundo	249	Waledac	34
Silly	489	Almanahe	241	Ackantta	32
Fakeav	489	Sasfis	199	Mebroot	26
Zbot	459	Graybird	166	Hotbar	21
Banker	449	Gammima	126	Qakbot	17
Virut	361	Mabezat	107		

Table 3.3: Malware families of the reference data set

3.7.1 Effectiveness of Unpacking Engine

Since the accuracy of any static malware analysis system depends on its capability of inspecting the original binary codes, unpacking is an important prerequisite for MutantX. To understand and confirm the popularity of malware writers' use of packers, we scanned the malware samples in the reference data set with PEiD [51], a popular packer detection tool that contains a large packer signature database, to detect packed samples. Our results show that about 30% (1470) of 4821 samples are packed with some type of packer. This percentage is smaller than the previously-reported number in [72]. A possible reason for this difference is that the latest version of PEiD (the one used in our experiment) is built in October 2008 while most of malware samples in the reference set are collected in April 2009. Since 10–15 new packers are created every month [72], many samples are likely to have been packed with packers that are unknown to PEiD. This also indicates the need and the benefit of generic unpacking techniques used in MutantX that can handle unknown or customized packers as well as multi-layered packers.

To evaluate the effectiveness of MutantX's unpacking, we select an unpacked mal-

ware sample (Deborm.ab) and packed it with 8 different packers. We then unpack all the packed binaries with `MutantX` and compare them with the original version. Ideally, the unpacked binary should be byte-to-byte identical to the original file. However, this is neither possible—`MutantX` does not reconstruct the import table, and the unpacker routine will also be dumped from the memory—nor necessary for the purpose of malware clustering. As a result, we compared the unpacked binaries with the original using two metrics, (i) the difference in the *instruction count* (IC) and (ii) the distance between N-gram feature vectors (NG) to assess the unpacking effectiveness, as they are directly related to the clustering accuracy. These comparison results are summarized in Table 3.4. For most packers, the `MutantX`'s unpacking engine successfully recovered their original binaries with only a 1–6% increase of ICs which is due to the inclusion of unpacker routines in the dumped memory. Besides, the feature vectors of unpacked binaries are very similar to that of the original binary with most distance measurements below 0.1, where distance 0 (1) means identical (completely different). However, `MutantX` also failed to unpack certain samples. In particular, we found that the memory dump of an Armadillo-packed malware sample still contains a packed version of the binary including unpacking code and encrypted payloads. A further investigation showed that Armadillo works by unpacking an intermediate executable on disk and creating another process to run this executable [72]. Therefore, dumping memory of an Armadillo-packed file does not capture the unpacked instructions. While running `MutantX` on a large data set, we have also observed other causes of unsuccessful unpacking, such as malware samples that do not run in a virtual machine or the time required for unpacking is longer than the threshold used in `MutantX` for the purpose of scalability. Despite these rare limitations, the generic unpacking technique used in `MutantX` is found very effective and able to handle many popular packers without requiring any specialized unpacking algorithm.

Packer	%diff in IC	dist of NG
ASprotect	6.70%	0.133
EXECryptor	3.20%	0.176
EXEStealth	0.88%	0.071
NSPack	0.87%	0.069
PEcompact	0.88%	0.068
UPX	0.88%	0.068
VMprotect	2.50%	0.100
Armadillo	–	–

Table 3.4: Unpacking effectiveness (IC: Instruction Count; NG: N-gram)

3.7.2 Malware Clustering Accuracy

We first evaluate and calibrate `MutantX` against the reference data set. The clustering component of `MutantX` is implemented based on the malheur package [88] that implements a prototype-based algorithm for clustering dynamic behavior. We modified malheur such that it takes as input disassembled instructions of a set of malware samples, converts them to a feature vector format (after applying the hash kernel), and runs a prototype-based clustering algorithm on the feature vectors. The output is a set of clusters $C = \{C_1, C_2, \ldots, C_c\}$. All of our evaluations were done on a Ubuntu 10.4 machine with the 2.6.35 linux kernel (Core i7 3.0G CPU and 12GB memory).

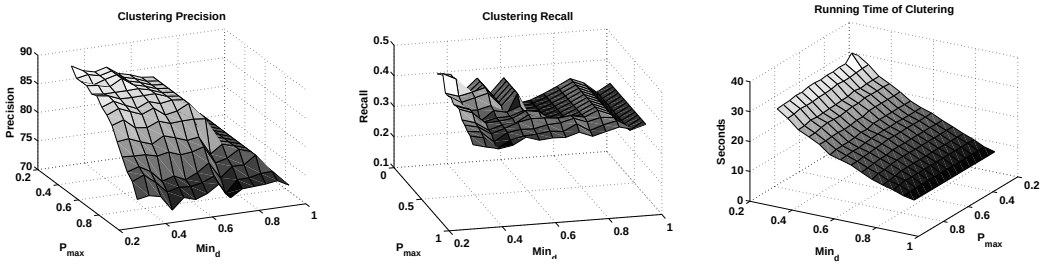


Figure 3.5: Precision, recall and running time of mutantX's clustering

We use *precision* and *recall* as the main metrics to assess the accuracy of MutnatX's clustering. Suppose that with respect to the original labels (i.e., family names), n input malware samples can be grouped into a set of clusters $O = \{O_1, O_2, \ldots, O_o\}$. Then, for a set C of clusters produced by `MutantX`'s clustering, precision P measures how well the individual clusters agree with the original malware classes (i.e.,

the exactness of clusters), and recall R measures how much the malware classes are scattered across the clusters (i.e., the completeness of each cluster). Formally, the precision P is defined as:

$$P = \frac{1}{n} \sum_{i=1} cmax(|C_i \cap O_1|, |C_i \cap O_2|, \ldots, |C_i \cap O_o|)$$

, and the recall R is defined as:

$$R = \frac{1}{n} \sum_{j=1} omax(|O_j \cap C_1|, |O_j \cap C_2|, \ldots, |O_i \cap C_c|)$$

. The precision will be 1 if all the samples in every cluster C_i are from the same family and the recall value will be 1 if all malware samples from the same family fall into a single cluster (but not necessarily the only family in this cluster). Figure 3.5 shows the precision and recall of `MutantX`'s clustering with varying thresholds P_{max} and Min_d. The number of N-grams used in the experiment is 4 and the number of hash bits is 12 (i.e., hash kernel maps the original features into 2^12 hash bins).

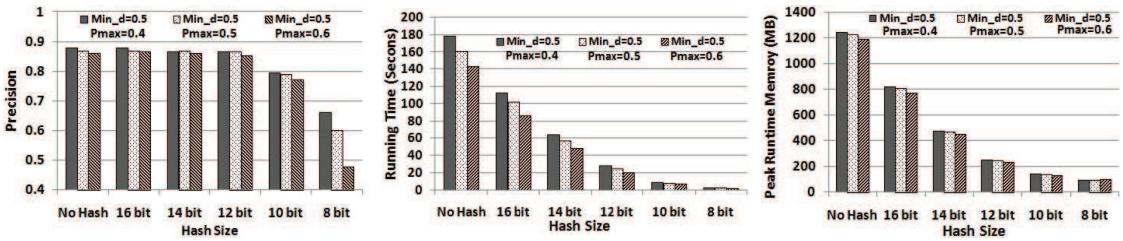
From the figure, we observe that `MutantX` is able to cluster the samples with the precision ranging from 0.72% to 0.89 (average=0.80). The precision number is smaller than those reported in previous approaches using dynamic behavior, e.g., 0.996 in [88] and 0.984 in [13]. While this difference may be due to different malware sets (and possibly incorrect labeling) used in our experiments, we conjecture that the reason for the higher accuracy of dynamic behavior approaches is more likely due to its resilience to low-level modifications such as packing and obfuscation, with the cost of longer running time and limited coverage. Therefore, the goal of `MutantX` is to provide an alternative way of categorizing malware that is complementary to the behavior-based analysis with better scalability while maintaining reasonably good accuracy. Indeed, Figure 3.5 shows that it takes only less than 30 seconds to complete the clustering for the entire reference dataset. In addition, we observe that the recall of

`MutantX` is around 0.3 and 0.4, which seems quite low. However, this low value of recall is expected, because we observe that there often exists a lot of diversity across malware variants. For instance some variant in a family can be several times larger in size than other variants in the same family. Possible reasons includes mislabeled samples, unidentified packers or heavily obfuscated binaries. Because of the highly diverse variants, `MutantX` tends to break the original family into several sub-families, resulting in a low recall value. For instance, `MutantX` creates more than 50 clusters for the reference dataset which contains 20 families according to the labels. Although less ideal, the breakdown into small subfamily is acceptable in practice, because the small families can be still useful in predicting the unknown sample's labels. Another observation from these results is that P_{max} (the threshold for distances from all data points to their nearest prototypes) has a greater influence on the clustering time since a smaller P_{max} value forces the algorithm to find more prototypes to cover all the data points, thus requiring a larger computation time. On the other hand, Min_d has a major impact on the clustering precision, i.e., increasing Min_d reduces the precision. The reason for this is that a smaller inter-cluster distance threshold will stop the prototype merging process earlier which will, in turn, reduce the probability of combining unrelated prototypes into a larger cluster. However, the price for this is the overfitting of clustering, i.e., the algorithm tends to create several small clusters that are less useful in determining the labels for unknown malware samples.

3.7.3 Validity of the Hashing Trick

We now evaluate the validity of the hash kernel in terms of its impact on the clustering speed and accuracy. A major concern in using the hashing trick in malware clusering is the possible loss of information due to the compression of high dimensional features into a lower dimensional space. Possible collisions between features may limit their overall expressiveness and may negatively impact the clustering. To

evaluate the efficacy of hashing trick in the clustering, we apply different hash sizes (i.e., the number of hash bins) on the original feature vector and study the resulting effect on the clustering accuracy. The hash function used in `MutantX` is MurmurHash 2.0 [4] which is a simple hash implementation with uniform value distribution, high throughput, and good colistin resistance. In order to gain a better understanding of the hashing trick performance, we also ran the clustering algorithm on the original feature vector without any compression, which serves as the baseline and best-possible case for assessing the effect of the hashing trick.



(a) (b) (c)

Figure 3.6: Precision, clustering time, and peak memory requirements with the number of hash bins ranging from 2^8 to 2^{16}, and without using the hashing trick .

Figure 3.6 compares the effects of hashing trick with different hash sizes (the number of hash bins ranging from 2^8 to 2^{16} and no hash) using three metrics—precision, clustering time and peak memory requirements. In Figure 3.6, different bars represent the results generated by using different parameter combinations all of which exhibit a similar trend when the hash size changes. From Figure 3.6, we find that as the hash size increases, the precision also improves because the collision probability decreases as there are more hash bins to hold the features. In fact, when the hash size is large enough, the probability of collision becomes negligible and the hashed features perform the same as the original features. For instance, in Figure 3.6, when more than 2^{12} hash bins are used, the clustering achieves almost the same precision (0.864 for 2^{12} hash bins) as the original features $P = 0.868$). In contrast, as the hash

size decreases, the impact of collision starts to surface. In particular, as the number of hash bins reduces to $2^8 = 256$, the precision decreases significantly and drops to below 0.5 for some parameter combinations. This is due probably to the collision of many critical features (e.g., features indicative of different families are now mapped to the same hash bin) that gives out conflicting signals and lowers the clustering precision. In this regard, a larger hash size is preferable, which adversely affects the clustering efficiency as shown in Figures 3.6 (b) and (c). These figures show that a small hash size is very effective in reducing the running time and memory footprints of the clustering algorithm, because each feature vector is smaller, thus requiring less memory for storage and less CPU cycles for computing the distance. For instance, as the hash size decreases from 16 bits to 8 bits, the required running time drops from almost 2 minutes to less than 10 seconds and the memory requirement decreases from 800 Mbytes to less than 100 Mbytes, at the cost of precision. As a result, `MutantX` has to make a tradeoff between accuracy and efficiency. In general, a 12-bit hash function is found to be a good compromise, reducing the time and memory requirements by over 80% while still providing good clustering accuracy. So, unless specified otherwise, all the experiments are performed with a 12-bit hash function. Figure 3.6 (c) also implies that as the number of malware samples increases, the hashing trick becomes critical. Without it, the memory requirement could become prohibitively high and very quickly (without hashing, clustering less than 5000 samples already requires more than 1.2 Gbytes of memory).

3.7.4 Impact of N-gram

Next, we assess the impact of N (of N-gram) on the clustering performance. Intuitively, features based on a larger N is more specific than those represented by a smaller N, because more instructions can be represented by each gram, providing better distinguishability between different samples. However, this comes at the cost

of exponential increase in the number of dimensions of the resulting feature vectors, because if there are m different grams, an N-gram feature vector will have m^N dimensions. Therefore, as N increases, the required storage and computation time could become very large. For instance, all 3-gram feature vectors (without hashing) for the reference malware set take up 289 Mbyte space and the number rises to 1.1 Gbytes when $N = 6$. Therefore, N has commonly been chosen to be small (2 or 3, and maximum 5). Fortunately, using the hashing trick enables us to compress the feature vectors and evaluate the case of using a large N. Our results are summarized in Figure 3.7 where 3-, 4-, 5-, 6-gram feature vectors are extracted from the reference data set, hashed by the same 12-bit hash function and clustered to assess their impact on the precision of clustering.

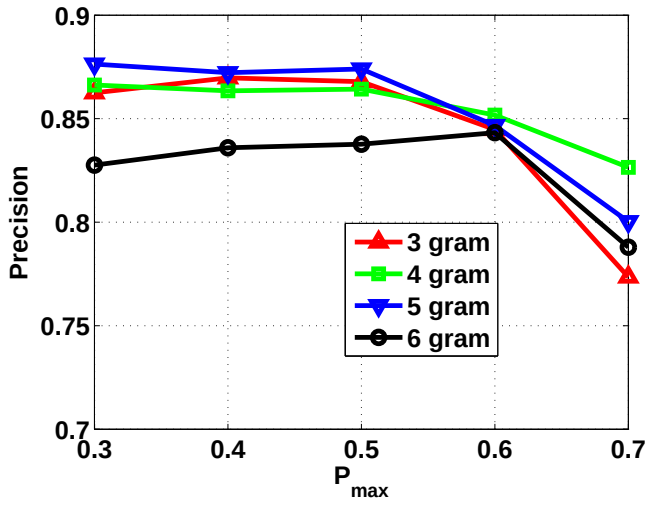


Figure 3.7: Precision of clustering with different N values .

From Figure 3.7, one can observe that use of a larger N value indeed improves the precision, e.g., 4- and 5-grams achieve better precision than 3-gram since larger grams can better capture the underlying semantics of the program and provide better distinguishability. However, it may not be apparent from the figure why 6-gram performs worst among them. One possible reason for this is that the number of features in the

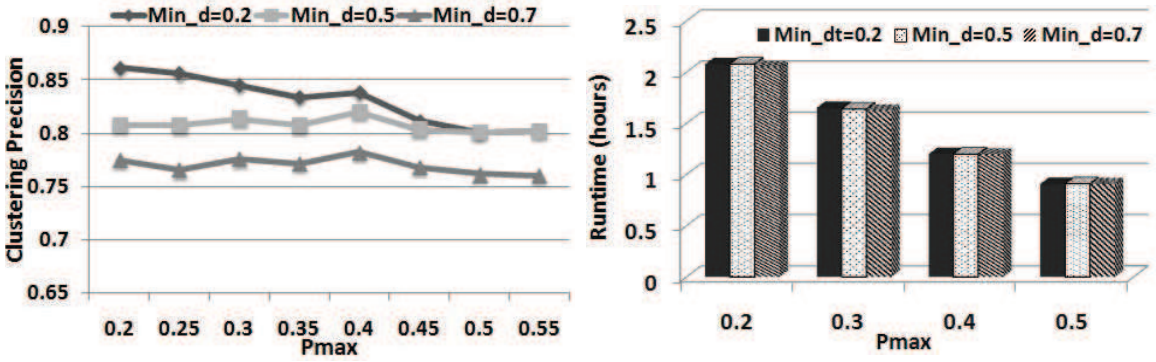
6-gram feature vector (i.e., m^6) is too large for the 12-bit hash function to handle. A large number of collisions between irrelevant features occur when hashed into a lower dimensional feature space, thus reducing the overall precision of clustering. In `MutantX`, we have chosen 4-gram, because even though 5-gram seems to provide better precision, the improvement is not large enough to warrant the additional storage and computation overheads.

3.7.5 Scalability of `MutantX`

Finally, we evaluate the scalability and accuracy of `MutantX` on a large malware data set with over 130,000 samples. According to [110], the labels are generated by applying Kaspersky AV software v5.5.18/Linux. We ran `MutantX` on the entire set of malware files with different combinations of parameters and plotted the results in Figure 3.8. Figure 3.8 (b) shows the amount of time for clustering the entire data set and the value P_{max} is shown to have a more significant impact on the running time. For example, when P_{max} is set to 0.5, the clustering takes less than 1 hour which is almost half of the time when P_{max} is set to 0.2. As we explained before, P_{max} determines the number of prototypes extracted from the input data which, in turn, determines the total number of distance computations required for clustering. However, as shown in Figure 3.8 (a), increasing P_{max} degrades the clustering precision. In certain cases (e.g., $Min_d = 0.2$), increasing P_{max} from 0.2 to 0.5 reduces the precision by almost 10%. This is not surprising because a large P_{max} allows each prototype to cover a large portion of the space, thus risking the possibility of including samples from irrelevant families. In general, with a reasonable setting (e.g., $Min_d = 0.5$ and $P_max = 0.4$), `MutantX` is able to complete the clustering in less than 1.5 hours with the precision close to 0.82.[1] The peak memory usage is around 3.6 Gbytes. These results indicate that `MutantX` is very efficient in handling a large number of samples

[1]The recall for the large data set is around 0.25 because of breaking the samples from large malware families into relatively small groups.

and thus has the potential to keep up with the huge influx of malware variants received nowadays.



(a) (b)

Figure 3.8: Precision, recall and running time of `MutantX`'s clustering for large number of malware programs

3.8 Limitations and Improvements

We now discuss limitations of the current `MutantX` prototype that could be exploited by adversaries to degrade its clustering effectiveness. As a static-analysis approach, `MutantX` is vulnerable to standard binary/instruction-level obfuscation. First, run-time packers and protectors still remain one of the biggest obstacles. Even though `MutantX` uses a generic unpacking algorithm, it is less effective against advanced packers, such as Armadillo, Themida and Xtreme-Protector, that employ very sophisticated protection mechanisms, e.g., driver-level protection, anti-debug, anti-emulation, etc. Although specialized unpacking tools that have been developed for these packers can be incorporated into `MutantX` to combat such packing, their constant evolution and customization escalate arms race between malware authors and defenders. Second, `MutantX` extracts features from disassembled malware code. Unfortunately, producing correct disassembly is often very challenging due to variable length instruction sets, mixture of code and data, indirect control flow that can be

exploited to confuse a disassembler. We have observed that a non-trivial number of malware samples obscure their binaries with various anti-disassembly tricks [117], such as making an infeasible conditional jump to the middle of next instruction (because most disassemblers statically follow the control flow to discover and disassemble the next instruction, this technique is very effective in confusing many disassemblers). Although the current `MutantX` prototype does not handle this type of obfuscation for simplicity, there are a variety of techniques [61] that can be used to mitigate this problem. Third, `MutantX` relies on the similarity of code instructions to cluster malware samples. It is possible to create syntactically distinct but semantically similar variants through instruction-level obfuscation, for example, by substituting equivalent instructions (e.g., changing every "mov r/m1, r/m2" to "push r/m2; pop r/m1"). One way to address these problems is to incorporate more advanced de-obfuscation techniques [85, 104] to normalize the malware code and maximize the similarity between malware variants. Note that dynamic-behavior-based approaches do not suffer from these limitations, but they come with their own deficiencies—limited coverage and scalability. Therefore, the goal of `MutantX` is not to replace the dynamic-behavior-based approaches. These two types of approaches should rather be combined to exploit their respective strengths and mitigate their weaknesses. Finally, `MutantX` cannot handle file infector or parasitic malware types, such as Sality Virus, which inject themselves into host executables. This is actually a general limitation for both static and dynamic behavior-based clustering. Since the size of a virus and its behavior traces could be relatively small compared to the host executables, unless the malware variants infect the same executables, a large portion of the code instructions or dynamic behaviors will appear different because of the distinct nature of host executables. Such parasitic malware is a matter of our future inquiry.

3.9 Conclusion

In this chapter, we have presented the design, implementation and evaluation of a malware clustering system based on static features, called `MutantX`, that can accurately and efficiently group malware variants according to the similarity in their code instructions. `MutantX` converts each malware program into a compact but effective representation (i.e., a sequence of encoded opcdoes) and performs prototype-based clustering on the corresponding N-gram feature vectors. To efficiently handle low-level mutation and obfuscation commonly employed by malware writers to evade detection, `MutantX` incorporates a generic unpacking technique to maximize the capability of analyzing the malware's original instructions and encode each instruction with its opcode to provide an appropriate level of generalization. For the scalability of clustering, `MutantX` uses a combination of a hashing kernel that reduces the dimensionality of feature vectors and a close-to-linear time prototype-based clustering, both of which together focus on a small set of representative samples for fast data organization. Equipped with these techniques, `MutantX` is experimentally shown to be able to process more than 100,000 malware samples within a few hours. As a static-analysis approach, `MutantX` is expected to be very effective and can be combined with any existing dynamic-behavior-based system to provide the level of accuracy and coverage required to pace with the current malware sample submission rate.

CHAPTER IV

Hancock: Automatic Generation of String Signatures for Malware Detection

4.1 Introduction

Symantec's anti-malware response group receives malware samples submitted by its customers as well as its competitors, analyzes them, and creates signatures that could be used to identify future instances of them in the field. The number of unique malware samples that Symantec receives has grown exponentially in the recent years, because malware programs are increasingly more customized, targeted, and intentionally restricted in terms of distribution scope. For example, the total number of distinct malware samples that Symantec observes exceeds 1 million, which is more than the combined sum of those of the previous years.

Although less proactive than desired, signature-based malware scanning is still the dominant approach to identifying malware samples in the wild because of its extremely low false positive rate, i.e., the probability of mistaking a goodware program for a malware program is very low. For example, the false positive rate requirement for Symantec's anti-malware signatures is below 0.1%. The majority of signatures used in existing signature-based malware scanners are *hash* signatures, each of which is the results of taking a hash function of a malware binary. Although hash signatures

have a low false positive rate, the number of malware samples covered by each hash signature is also low, typically one signature per malware sample. As a result, the total size of the hash signatures required also grows with the exponential growth in the number of unique malware samples. This creates a signature distribution problem for Symantec: How to distribute these hash-based malware signatures to hundreds of millions of users across the world several dozen times per day in a scalable way?

One possible solution to the signature explosion problem is to replace hash signatures with *string* signatures, each of which corresponds to a contiguous byte sequence appearing in a malware binary. Traditionally, string signatures are created manually because it is difficult to automatically determine which byte sequence in a malware binary is less false positive (FP)-prone, i.e., unlikely to appear in any goodware program in the world. Even for the manually created string signatures, it is generally straightforward for malware authors to evade them, because they typically correspond to easy-to-modify data strings in malware binaries, for example, names of malware authors, special pop-up messages, etc. `Hancock` is an automatic string signature generation system developed in Symantec Research Labs that aims to automate the process of creating high-quality string signatures that simultaneously minimizes the false positive rate and maximizes the malware coverage. That is, the probability that a `Hancock`-generated string signature appears in any goodware program should be very very low, and at the same time each `Hancock`-generated string signature can be used to identify as many malware programs as possible.

Given a set of malware samples, `Hancock` is designed to create a minimal set of N-byte sequences each of which has a sufficiently low false positive rate, and that collectively cover as large a portion of the malware set as possible. Based on previous empirical studies, `Hancock` sets N to 48. `Hancock` examines every 48-byte sequence in the input malware set, and filters out those byte sequences whose estimated occurrence probability in goodware programs according to a pre-computed goodware

model is above a certain threshold, that are considered a part of library functions, or that are not sufficiently interesting or unique based on a set of heuristic rules that encode malware analysts' selection criteria. Among those signature candidates that pass the initial filtering step, `Hancock` further applies a set of selection rules based on the *diversity* principle: If the set of malware samples containing a signature candidate are more similar to one another, the less FP-prone is the signature candidate. Finally, `Hancock` is extended to generate string signatures that consist of multiple disjoint byte sequences rather than only one contiguous byte sequence. Although multi-component string signatures are more effective than single-component signatures, they also incur higher run-time performance overhead because individual components are more likely to match goodware programs. In the following sections, we will describe the signature filter algorithms, the signature selection algorithms, and the multi-component generalization used in `Hancock`.

4.2 Related Work

Modern anti-virus software typically employ a variety of methods to detect malware programs, such as signature-based scanning [23], heuristic-based detection [5] and behavioral detection [40]. Although less proactive, signature-based malware scanning is still the most prevalent approach to identify malware because of its efficiency and low false positive rate. Traditionally, the malware signatures are created manually, which is both slow and error-prone. As a result, efficient generation of malware signatures has become one major challenge for Anti-virus companies to handle the significantly increasing number of new malware threats. To solve this problem, several automatic signature generation approaches have been proposed.

Most previous work focused on creating signatures that are used by Network Intrusion Detection Systems (NIDS) to detect network worms. Singh et al. proposed EarlyBird [94], which used packet content prevalence and address dispersion to au-

tomatically generate worm signatures from the invariant portions of worm payloads. Autograph [2] exploited a similar idea to create worm signatures by dividing each suspicious network flow into blocks terminated by some breakmark and then analyzing the prevalence of each content block. The suspicious flows are selected by a port-scanning flow classifier to reduce false positives. Kreibich and Crowcroft developed Honeycomb [59], a system that used honeypots to gather inherently suspicious traffic and generates signatures by applying the longest common substring (LCS) algorithm to search for similarities in the packet payloads. One potential drawback of signatures generated from previous approaches is that they are all continuous strings and may fail to match polymorphic worm payloads. Polygraph [76], instead, searched for invariant contents in the network flows and created signatures consisting of multiple disjoint content substrings. Polygraph also utilized a naive Bayes classifier to allow the probabilistic matching and classification, and thus provided better proactive detection capabilities. Li et al proposed Hasma [65], a system that used a model-based algorithm to analyze the invariant contents of polymorphic worms and analytically prove the attack-resilience of generated signatures. PDAS (Position-Aware Distribution Signatures) [98] took advantage of a statistical anomaly-based approach to improve the resilience of signatures to polymorphic malware variants. Another common method for detecting polymorphic malware is to incorporate semantics-awareness into signatures. For example, Christodorescu *et al.* proposed static semantics-aware malware detection in [73]. They applied a matching algorithm on the disassembled binaries to find the instruction sequences that match the manually generated templates of malicious behaviors, e.g., decryption loop. Yegneswaran et.al developed Nemean [119], a framework for automatic generation of intrusion signatures from honeynet packet traces. Nemean applied clustering techniques on connections and sessions to create protocol-semantic-aware signatures, thereby reducing the possibility of false alarms.

Another loosely related area is the automatic generation of attack signatures, vulnerability signatures and software patches. TaintCheck [77] and Vigilante [25] applied taint analysis to track the propagation of network inputs to data used in attacks, e.g., jump addresses, format strings and system call arguments, which are used to create signatures for the attacks. Other heuristic-based approaches [66, 67, 112, 114] have also been proposed to exploit properties of specific exploits (e.g., buffer overflow) and create attack signatures. Generalizing from these approaches, Brumley et al. proposed a systematic method [19] that used a formal model to reason about vulnerability signatures and quantify the signature qualities. An alternative approach to preventing malware from exploiting vulnerabilities is to apply data patches (e.g. Shield vulnerability signatures [111]) in the firewalls to filter malicious traffic. To automatically generate data patches, Cui et al. proposed ShieldGen [26], which leveraged the knowledge of data format of malicious attacks to generate potential attack instances and then created signatures from the instances that successfully exploit the vulnerabilities.

`Hancock` differs from previous work by focusing on automatically generating high-quality string signatures with extremely low false positives. Our research was based loosely on the virus signature extraction work [54] by Kephart and Arnold, which was commercially used by IBM. In their work, the researchers used a 5-gram Markov chain model of good software to estimate the probability that a given byte sequence would show up in good software. They tested published, hand-generated signatures and found that it was quite easy to set a model probability threshold with a zero false positive rate and a modest false negative rate (the fraction of rejected signatures that would not be found in goodware) of 48%.

Symantec acquired this technology from IBM in the mid-90s and found that it led to many false positives. The Symantec engineers believed that it worked well for IBM because IBM's anti-virus technology was used mainly in corporate environments,

making it much easier for IBM to collect a representative set of goodware. By contrast, signatures generated by `Hancock` are mainly for home users who use a much broader set of goodware. The model's training set cannot possibly contain, or even represent, all of this goodware. This poses a significant challenge for `Hancock` in avoiding FP-prone signatures.

4.3 Signature Candidate Selection

4.3.1 Goodware Modeling

The first line of defense in `Hancock` is a Markov chain-based model that is trained on a large goodware set and is designed to estimate the probability of a given byte sequence appearing in goodware. If the probability of a candidate signature appearing in some goodware program is higher than a threshold, `Hancock` rejects it. Compared with standard Markov models, `Hancock`'s goodware model has two important features:

- **Scalable to very large goodware set** Symantec regularly tests its anti-virus signatures against several terabytes of goodware programs. A standard Markov model uses $O(N)$ space [90], where N is the number of bytes in the training set. `Hancock`'s goodware model focuses only on high-information-density byte sequences so as to scale to very large goodware training sets.

- **Focusing on rare byte sequences** For a candidate signature not to cause a false positive, its probability of appearing in goodware must be very, very low. Therefore, the primary goal of `Hancock`'s model is to distinguish between low-probability byte sequences and rare byte sequences.

4.3.1.1 Basic Algorithm

The model used in `Hancock` is a fixed-order 5-gram Markov chain model, which estimates the probability of the fifth byte conditioned on the occurrence of the preced-

ing four bytes. Training consists of counting instances of 5-grams – 5-byte sequences – as well as 4-grams, 3-grams, etc. The model calculates the probability of a 48-byte sequence by multiplying estimated probabilities of each of the 48 bytes. A single byte's probability is the probability of that byte following the four preceding bytes. For example, the probability that "e" follows "abcd" is

$$p(\text{e}|\text{abcd}) \quad = \quad \frac{count(\text{abcde})}{count(\text{abcd})} * (1 - \epsilon(count(\text{abcd}))) + p(\text{e}|\text{bcd}) * \epsilon(count(\text{abcd}))$$

In this equation, $count(s)$ is the number of occurrences of the byte sequence s in the training set. $\epsilon(count(s))$ is the *escape mass* of s, which limits overtraining. Escape mass decreases with count. Empirically, we found that a good escape mass for our model is $\epsilon(c) = \frac{\sqrt{32}}{\sqrt{32}+\sqrt{c}}$.

4.3.1.2 Model Pruning

The memory required for a vanilla fixed-order 5-gram model is significantly greater than the size of the original training set. `Hancock` reduces the memory requirement of the model by incorporating an algorithm that prunes away less useful grams in the model. The algorithm looks at the *relative information gain* of a gram and eliminates it if its information gain is too low. This allows `Hancock` to keep the most valuable grams given a fixed memory constraint.

Consider a model's grams viewed as nodes in a tree. The algorithm moves considers every node X, corresponding to byte sequence s, whose children (corresponding to $s\sigma$ for some byte σ) are all leaves. Let s' be s with its first byte removed. For example, if s is "abcd", s' is "bcd". For each child of X, σ, the algorithm compares $p(\sigma|s)$ to $p(\sigma|s')$. In this example, the algorithm compares $p(\text{a}|\text{abcd})$ to $p(\text{a}|\text{bcd})$, $p(\text{b}|\text{abcd})$ to $p(\text{b}|\text{bcd})$, etc. If the difference between $p(\sigma|s)$ and $p(\sigma|s')$ is smaller than a threshold, that means that X is does not add that much value to σ and the

node σ can be pruned away without compromising the model's accuracy.

To focus on low-probability sequences, `Hancock` uses the difference between the logs of these two probabilities, rather than that between their raw probability values. Given a space budget, `Hancock` keeps adjusting the threshold until it hits the space target.

4.3.1.3 Model Merging

Creating a pruned model requires a large amount of intermediate memory, before the pruning step. Thus, the amount of available memory limits the size of the model that can be created. To get around this limit, `Hancock` creates several smaller models on subsets of the training data, prunes them, and then merges them.

Merging a model $M1$ with an existing model $M2$ is mostly a matter of adding up their gram counts. The challenge is in dealing with grams pruned from $M1$ that exist in $M2$ (and vice versa). The merging algorithm must recreate these gram counts in $M1$. Let $s\sigma$ be such a gram and let s' be s with its first byte removed. The algorithm estimates the count for $s\sigma$ as $count(s) * p(\sigma|s')$. Once these pruned grams are reconstituted, the algorithm simply adds the two models' gram counts.

4.3.1.4 Experimental Results

We created an occurrence probability model from a 1-GByte training goodware set and computed the probability of a large number of 24-byte test sequences, extracted from malware files. We checked each test byte sequence against a goodware database, which is a superset of the training set, to determine if it is a true positive (a good signature) or a false positive (which occurs in goodware). In Figure 4.1, each point in the FP and TP curves represents the fraction (Y axis value) of test byte sequences whose model probability is below the X axis value.

As expected, TP signatures have much lower probabilities, on average, than FP

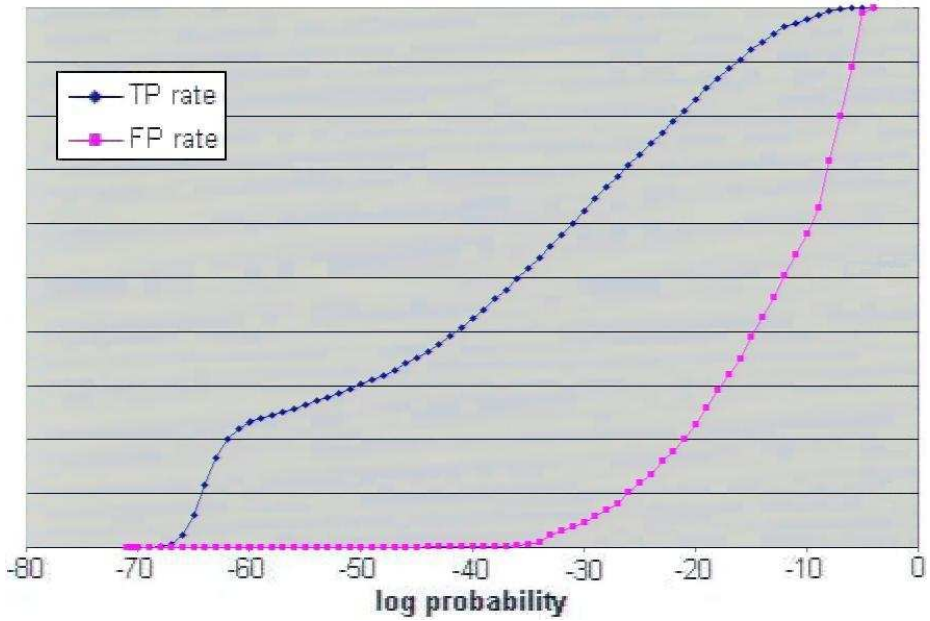


Figure 4.1: The fractions of false positive and true positive test sequences with occurrence probabilities below the X axis value

signatures. A small number of FP signatures have very low probabilities – below 10^{-60}. Around probability 10^{-40}, however, the model does provide excellent discrimination power, rejecting 99% of FP signatures and accepting almost half of TP signatures.

To evaluate the effectiveness of `Hancock`'s information gain-based pruning algorithm, we used two sets of models: non-pruned and pruned. The former were trained on 50 to 100 Mbytes of goodware. The latter were trained on 100 Mbytes of goodware and pruned to various sizes. For each model, we then computed its TP rate at the probability threshold that yields a 2% FP rate. Figure 4.2 shows these TP rates of goodware models versus the model's size in memory. In this case, pruning can roughly halve the goodware model size while offering the same TP rate as the pruned model derived from the same training set.

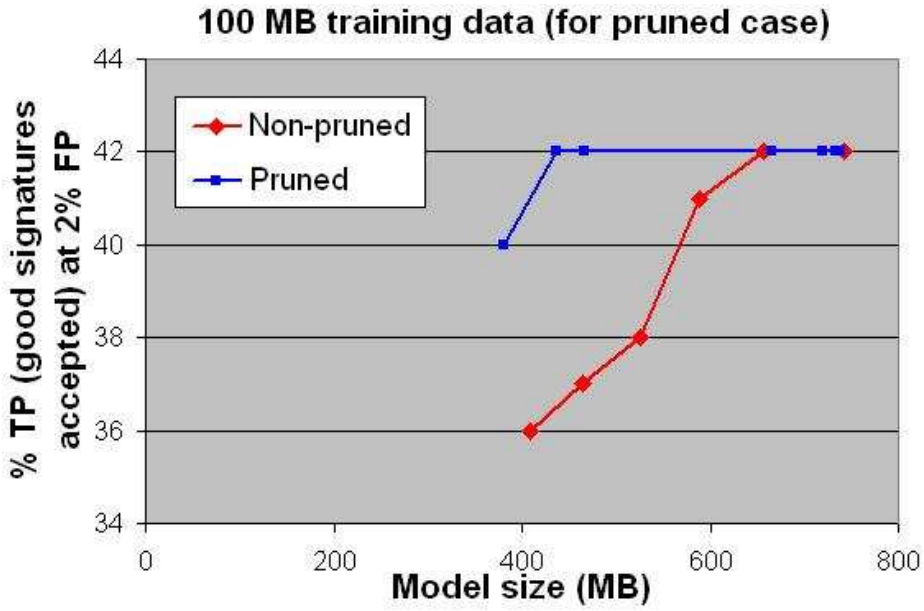


Figure 4.2: TP rate comparison between pruned models and non-pruned models when the training set varies from 50 Mbytes to 100 Mbytes

4.3.2 Library Function Recognition

A library is a collection of standard functions that implement common operations, such as file IO, string manipulation and graphics. Modern malware authors use library functions extensively to simplify development, just like goodware authors. By construction, variants of a malware family are likely to share some library functions. Because these library functions also have a high probability of appearing in goodware, `Hancock` needs to remove them from consideration when generating string signatures. Toward this goal, we developed a set of library function recognition techniques to determine whether a function in a malware file is likely to be a library function or not.

A popular library identification technique is IDA Pro's Fast Library Identification and Recognition Technology (FLIRT) [48], which uses byte pattern matching (similar to string signature scanning) algorithms to quickly determine whether a disassembled function matches any of the signatures known to IDA Pro.[1] Although FLIRT is very

[1]IDA Pro ships with a database of signatures for about 120 libraries associated with common

accurate in pinpointing common library functions, it still needs some improvement to suit `Hancock`'s needs. First, FLIRT is designed to never falsely identify a library. To achieve this, FLIRT first tries to identify the compiler type (e.g., Visual C++ 7.0, 8.0, Borland C++, Delphi, etc.) of a disassembled program and applies only signatures for that compiler. For example, vcseh signatures (Structured Exception Handling library signatures) will only be applied to binary files that appear to have been compiled with Visual C++ 7 or 8. This conservative approach can lead to false negatives (a library function not identified) because of failure in correctly detecting the compiler type and/or lack of signatures for libraries. In addition, because FLIRT uses a rigorous pattern matching algorithm to search for signatures, small variation in libraries, e.g., minor changes in the source code, different settings in compiler optimization options or use of different compiler versions to build the library, could prevent FLIRT from recognizing all library functions in a disassembled program.

In contrast to FLIRT's conservative approach, `Hancock`'s primary goal is to eliminate false positive signatures. It takes a more aggressive stance by being willing to mistake non-library functions for library functions. Such misidentification is acceptable because it prevents any byte sequence that is potentially associated with a library function from being used as a malware signature. We exploited this additional latitude with the following three heuristics:

Universal FLIRT Heuristic This heuristic generalizes IDA Pro's FLIRT technique by matching a given function against all FLIRT signatures, regardless of whether they are associated with the compiler used to compile the function. This generalization is useful because malware authors often post-process their malware programs to hide or obfuscate compiler information in an attempt to deter any reverse engineering efforts. Moreover, any string signatures extracted from a function in a program compiled by a compiler C1 that looks like a library function in another compiler C2

compilers. Each signature corresponds to a binary pattern in a library function.

are likely to cause false positives against programs compiled by C2, and thus should be rejected.

Library Function Reference Heuristic This heuristic identifies a library function if the function is statically called, directly or indirectly, by any known library function. The rationale behind this heuristic is that since a library cannot know in advance which user program it will be linked to, it is impossible for a library function to statically call any user-written function, except callback functions, which are implemented through function pointers and the call is dynamically resolved. As a result, it is safe to mark all children of a library function in its call tree as library functions. Specifically, the proposed technique disassembles a binary program, builds a function call graph representation of the program, and marks any function that is called by a known library function as a library function. This marking process repeats itself until no new library function can be found.

In general, compilers automatically include into an executable binary certain template code, e.g., startup functions, error handling, etc., which IDA Pro also considers as library functions as well. These template functions and their callees must be excluded in the above library function marking algorithm. For example, the entry point function *start* and *mainCRTstartup* in Visual C++-compiled binaries are created by the compiler to perform startup preparation (e.g., execute global constructors, catch all uncaught exceptions) before invoking the user-defined main function.

Address Space Heuristic This heuristic identifies a library function based on whether its neighboring functions in the binary file are library functions. When a library is statically linked into a program, the binary codes of the library usually occupy a contiguous address space range and sometimes there is padding space between adjacent functions. Therefore, to detect those internal, non-exported functions in a library that we cannot prove statically are called by some known library function, we

exploit the physical proximity property of functions that belong to the same library. More specifically, we mark a function as a library functions if:

- It is immediately surrounded or sandwiched by known library functions, and
- The size of the space between it and its neighboring library functions is below a certain threshold. In `Hancock`, we set the threshold to be 128 bytes, based on a statistical analysis of inter-library-function space in binary programs generated by commercial compilers.

In `Hancock`, we implement the above library function heuristics as an IDA Pro plugin. After IDA Pro disassembled a malware program, `Hancock` first applies the Universal FLIRT heuristic to maximize the detection capability with FLIRT's library signatures. Then the Function Reference and the Address Space heuristics are repeated until the set of identified library functions converges. Although these techniques collectively could mistake non-library functions for library functions, in practice such false positives do not have noticeable impacts on the effectiveness of `Hancock`'s overall signature generation capability. However, by eliminating up front more of the shared library code in binaries, we have found that we can significantly reduce the number of false positives signatures. In addition, these techniques also improve the signature quality, because it allows `Hancock` to focus more on their coverage than on their FP likelihood.

4.3.3 Code Interestingness Check

The *code interestingness* check is designed to capture the intuitions of Symantec's malware analysis experts about what makes a good string signature. For the most part, these metrics identify signatures that are less likely to be false positives. They can also identify malicious behavior, though avoiding false positives is the main goal. The code interestingness check assigns a score for each "interesting" instruction

pattern appearing in a candidate signature, sums up these scores, and rejects the candidate signature if its sum is below a threshold, i.e. not interesting enough. The interesting patterns used in `Hancock` are:

- **Unusual constant values** Constants sometimes have hard-coded values that are important to malware, such as the IP address and port of a command and control server. More importantly, if a signature has unusual constant values, it is less likely to be a false positive.

- **Unusual address offsets** Access to memory that is more than 32 bytes from the base pointer can indicate access to a large class or structure. If these structures are unique to a malware family, then accesses to particular offsets into this structure are less likely to show up in goodware. This pattern is not uncommon among legitimate Win32 applications. Nonetheless, it has good discrimination power.

- **Local or non-library function calls** A local function call itself is not very distinctive, but the setup for local function calls often is, in terms of how it is used and how its parameters are prepared. In contrast, setup for system calls is not as interesting, because they are used in many programs and invoked in a similar way.

- **Math instructions** A malware analyst at Symantec noted that malware often perform strange mathematical operations, to obfuscate and for various other reasons. Thus, `Hancock` looks for strange sequences of XORs, ADDs, etc. that are unlikely to show up in goodware.

4.4 Signature Candidate Filtering

`Hancock` selects candidate signatures using techniques that assess a candidate's FP probability based solely on its contents. In this section, we describe a set of filtering techniques that remove from further consideration those candidate signatures that are likely to cause a false positive based on the signatures' use in malware files.

`Hancock` is designed to generate signatures, each of which is capable of identifying variants of one or a small number of malware families. Therefore, the set of malware files covered by a `Hancock` signature should be similar to one another. Based on this insight, the general guiding principle behind `Hancock`'s signature candidate filtering mechanisms is to estimate how FP-prone a candidate string signature is based on how diverse the set of malware files covered by the string signature are. The more diverse these files are, the more likely the signature covering them is to appear in goodware programs. `Hancock` measures the diversity of a set of binary files based on their byte-level and instruction-level representations. The following two subsections describe these two diversity measurement methods.

4.4.1 Byte-Level Diversity

Given a signature S and the set of files it covers, X, `Hancock` measures the byte-level similarity or diversity among the files in X by extracting the byte-level context surrounding S and computing the similarity among these contexts. More concretely, `Hancock` employs the following four types of byte-level signature-containing contexts for diversity measurement.

Malware Group Ratio/Count `Hancock` clusters malware files into groups based on their byte-level histogram representation. It then counts the number of groups to which the files in X belong. If this number divided by the number of files in X exceeds a threshold ratio, or if the number exceeds a threshold count, `Hancock` rejects

S. These files cannot be variants of a single malware family, if each malware group indeed corresponds to a malware family.

Signature Position Deviation `Hancock` calculates the position of S within each file in X, and computes the standard deviation of S's positions in these files. If the standard deviation exceeds a threshold, `Hancock` rejects S, because a large positional deviation suggests that S is included in the files it covers for very different reasons. Therefore, these files are unlikely to belong to the same malware family. The position of S in a malware file can be an absolute byte offset, which is with respect to the beginning of the file, or a relative byte offset, which is with respect to the beginning of the code section containing S.

Multiple Common Signatures `Hancock` attempts to find another common signature that is present in all the files in X and is at least 1 Kbyte away from S. If such a common signature indeed exists and the distance between this signature and S has low standard deviation among the files in X, then `Hancock` accepts S because this suggests the files in X share a large chunk of code and thus are likely to be variants of a single malware family. Intuitively, this heuristic measures the similarity among files in X using additional signatures that are sufficiently far away, and can be generalized to using the third or fourth signature.

Surrounding Context Count `Hancock` expands S in each malware file in X by adding bytes to its beginning and end until the resulting byte sequences become different. For each such distinct byte sequence, `Hancock` repeats the same expansion procedure until the expanded byte sequences reach a size limit, or when the total number of distinct expanded byte sequences exceeds a threshold. If this expansion procedure terminates because the number of distinct expanded byte sequences exceeds a threshold, `Hancock` rejects S, because the fact that there are more than several dis-

tinct contexts surrounding S among the files in X suggests that these files do not belong to the same malware family.

4.4.2 Instruction-Level Diversity

Although byte-level diversity measurement techniques are easy to compute and quite effective in some cases, they treat bytes in a binary file as numerical values and do not consider their semantics. Given a signature S and the set of files it covers, X, instruction-level diversity measurement techniques, on the other hand, measure the instruction-level similarity or diversity among the files in X by extracting the instruction-level context surrounding S and computing the similarity among these contexts. More concretely, `Hancock` employs the following three different types of instruction-level signature-containing contexts for diversity measurement.

Enclosing Function Count `Hancock` extracts the enclosing function of S in each malware file in X, and counts the number of distinct enclosing functions. If the number of distinct enclosing functions of S with respect to X is higher than a threshold, `Hancock` rejects S, because S appears in too many distinct contexts among the files in X and therefore is not likely to be an intrinsic part of one or a very small number of malware families. To determine if two enclosing functions are distinct, `Hancock` uses the following three identicalness measures, in decreasing order of strictness:

- The byte sequences of the two enclosing functions are identical.
- The instruction op-code sequences of the two enclosing functions are identical. `Hancock` extracts the op-code part of every instruction in a function, and normalizes variants of the same op-code class into their canonical op-code. For example, there are about 10 different X86 op-codes for ADD, and `Hancock` translates all of them into the same op-code. Because each instruction's operands

are ignored, this measure is resistant to intentional or accidental polymorphic transformations such as re-locationing, register assignment, etc.

- The instruction op-code sequences of the two enclosing functions are identical after *instruction sequence normalization*. Before comparing two op-code sequences, `Hancock` performs a set of de-obfuscating normalizations that are designed to undo simple obfuscating transformations, such as replacing `test esi, esi` with `or esi, esi`, replacing `push ebp; mov ebp, esp` with `push ebp; push esp; pop ebp`, etc.

Enclosing Subgraph Count `Hancock` extracts a subgraph of the function call graph of every file in X centered at the call graph node corresponding to S's enclosing function, and compares the number of distinct enclosing subgraphs surrounding S. An enclosing subgraph is N-level if it contains up to N-hop neighbors of S's enclosing function in the function call graph. In practice, N is set to either 1 or 2. If the number of distinct N-level enclosing subgraphs of S with respect to X is higher than a threshold, `Hancock` rejects S, because S is not likely to be an intrinsic part of one or a very small number of malware families. This approach defines the surrounding context of S based on the set of functions that directly or indirectly call or are called by S's enclosing function, and their calling relationships. To abstract the body of the functions in the enclosing subgraphs, `Hancock` labels each node as follows: (1) If a node corresponds to a library function, `Hancock` uses the library function's name as its label. (2) If a node corresponds to a non-library function, `Hancock` labels it with the sequence of known API calls in the corresponding function. After labeling the nodes, `Hancock` considers two enclosing subgraphs as distinct if the edit distance between them is above a certain threshold.

Call Graph Cluster Count `Hancock` extracts the complete function call graph associated with every file in X, where each call graph node corresponds to either a

non-library function or an entry-point library function, and partitions these graphs into different clusters according to their topological structure. Functions internal to a library are ignored. Nodes are labeled in the same way as described above. The assumption here is that the function call graphs of variants of a malware family are similar to one another, but the function call graphs of different malware families are distinct from one another. The distance threshold used in graph clustering is adaptively determined based on the average size of the input graphs. If the number of clusters obtained this way is higher than a threshold, `Hancock` rejects S because S seems to covers too many malware families and is thus likely to be an FP.

In summary, the byte-level counterparts of the enclosing function count, the enclosing subgraph count and the call graph cluster count are the surrounding context count, the deviation in signature position, and malware group count, respectively. The byte-level multiple common signature heuristic is a sampling technique to determine if the set of malware files covering a signature share a common context surrounding the signature.

4.5 Multi-Component String Signature Generation

Traditionally, string signatures used in AV scanners consist of a contiguous sequence of bytes. We refer to these as single-component signature (SCS). A natural generalization of SCS is multi-component signatures (MCS), which consist of multiple byte sequences that are potentially disjoint from one another. For example, we can use a 48-byte SCS to identify a malware program; for the same amount of storage space, we can create a two-component MCS with two 24-byte sequences. Obviously, an N-byte SCS is a special case of a K-component MCS where each component is of size $\frac{N}{K}$. Therefore, given a fixed storage space budget, MCS provides more flexibility in choosing malware-identifying signatures than SCS, and is thus expected to be more effective in improving coverage without increasing the false positive rate.

In the most general form, the components of a MCS do not need to be of the same size. However, to limit the search space, in the `Hancock` project we explore only those MCSs that have equal-sized components. So the next question is how many components a MCS should have, given a fixed space budget. Intuitively, each component should be sufficiently long so that it is unlikely to match a random byte sequence in binary programs by accident. On the other hand, the larger the number of components in a MCS, the more effective it is in eliminating false positives. Given the above considerations and the practical signature size constraint, `Hancock` chooses the number of components in each MCS to be between 3 and 5.

`Hancock` generates the candidate component set using a goodware model and a goodware set. Unlike SCS, candidate components are drawn from both data and code, because intuitively, combinations of code component signatures and data component signatures make perfectly good MCS signatures. When `Hancock` examines an $\frac{N}{K}$-byte sequence, it finds the longest substring containing this sequence that is common to all malware files that have the sequence. `Hancock` takes only one candidate component from this substring. It eliminates all sequences that occur in the goodware set and then takes the sequence with the lowest model probability. Unlike SCS, there is no model probability threshold.

Given a set of qualified component signature candidates, S1, and the set of malware files that each component signature candidate covers, `Hancock` uses the following algorithm to arrive at the final subset of component signature candidates used to form MCSs, S2:

1. Compute for each component signature candidate in S1 its *effective coverage value*, which is a sum of weights associated with each file the component signature candidate covers. The weight of a covered file is equal to its *coverage count*, the number of candidates in S2 already covering it, except when the number of component signatures in S2 covering that file is larger than or equal to K, in

which case the weight is set to zero.

2. Move the component signature candidate with the highest effective coverage value from S1 to S2, and increment the coverage count of each file the component signature candidate covers.

3. If there are still malware files that are still uncovered or there exists at least one component signature in S1 whose effective coverage value is non-zero, go to Step 1; otherwise exit.

The above algorithm is a modified version of the standard greedy algorithm for the *set covering* problem. The only difference is that it gauges the value of each component signature candidate using its effective coverage value, which takes into account the fact that at least K component signatures in S2 must match a malware file before the file is considered covered. The way weights are assigned to partially covered files is meant to reflect the intuition that the value of a component signature candidate to a malware file is higher when it brings the file's coverage count from $X-1$ to X than that from $X-2$ to $X-1$, where X is less than or equal to K.

After S2 is determined, `Hancock` finalizes the K-component MCS for each malware file considered covered, i.e., whose coverage count is no smaller than K. To do so, `Hancock` first checks each component signature in S2 against a goodware database, and marks it as an FP if it matches some goodware file in the database. Then `Hancock` considers all possible K-component MCSs for each malware file and chooses the one with the smallest number of components that are an FP. If the number of FP components in the chosen MCS is higher than a threshold, T_{FP}, the MCS is deemed as unusable and the malware file is considered not covered. Empirically, T is chosen to be 1 or 2. After each malware file's MCS is determined, `Hancock` applies the same diversity principle to each MCS based on the malware files it covers.

Threshold setting	Model probability	Group ratio	Position deviation	# common signatures	Interes-tingness	Minimum coverage
Loose	-90	0.35	4000	1	13	3
Normal	-90	0.35	3000	1	14	4
Strict	-90	0.35	3000	2	17	4

Table 4.1: Heuristic threshold settings

4.6 Evaluation

4.6.1 Methodology

To evaluate the overall effectiveness of `Hancock`, we used it to generate 48-byte string signatures for two sets of malware files, and use the coverage and number of false positives of these signatures as the performance metrics. The first malware set has 2,363 unpacked files that Symantec gathered in August 2008. The other has 46,288 unpacked files gathered in 2007-2008. The goodware model used in initial signature candidate filtering is derived from a 31-Gbyte goodware training set. In addition, we used another 1.8-Gbyte goodware set to filter out FP-prone signature candidates. To determine which signatures are FPs, we tested each generated signature against a 213-Gbyte goodware set. The machine used to perform these experiments has four quad-core 1.98-GHz AMD Opteron processors and 128 Gbytes of RAM.

4.6.2 Single-Component Signatures

Because almost every signature candidate selection and filtering technique in `Hancock` comes with an empirical threshold parameter, it is impossible to present results corresponding to all possible combinations of these parameters. Instead, we present results corresponding to three representative settings, which are shown in Table 4.1 and called *Loose*, *Normal* and *Strict*. The generated signatures cover overlapping sets of malware files.

To gain additional assurance that `Hancock`'s FP rate was low enough, Symantec's

Threshold setting	Cover-age	# sig.s	# FPs	Good sig.s	Poor sig.s	Bad sig.s
Loose	15.7%	23	0	6	7	1
Normal	14.0%	18	0	6	2	0
Strict	11.7%	11	0	6	0	0

Table 4.2: Results for August 2008 data

malware analysts wanted to see not only zero false positives, but also that the signatures look good – they look like they encode non-generic behavior that is unlikely to show up in goodware. To that end, we manually ranked signatures on the August 2008 malware set as good, poor, and bad.

These results show not only that `Hancock` has a low false positive rate, but also that tighter thresholds can produce signatures that look less generic. Unfortunately, it can only produce signatures to cover a small fraction of the specified malware.

Several factors limit `Hancock`'s coverage:

- `Hancock`'s packer detection might be insufficient. PEiD recognizes many packers, but by no means all of them. Entropy detection can also be fooled: some packers do not compress the original file's data, but only obfuscate it. Diversity-based heuristics will probably reject most candidate signatures extracted from packed files. (Automatically generating signatures for packed files would be bad, anyway, since they would be signatures on packer code.)

- `Hancock` works best when the malware set has many malware families and many files in each malware family. It needs many families so that diversity-based heuristics can identify generic or rare library code that shows up in several malware families. It needs many files in each family so that diversity-based heuristics can identify which candidate signatures really are characteristic of a malware family. If the malware sets have many malware families with only a few files each, this would lower `Hancock`'s coverage.

Threshold setting	Coverage	# sig.s	# FPs
Loose	14.1%	1650	7
Normal	11.7%	767	2
Normal, pos. deviation 1000	11.3%	715	0
Strict	4.4%	206	0

Table 4.3: Results for 2007-8 data

- Malware polymorphism hampers `Hancock`'s effectiveness. If only some code is polymorphic, `Hancock` can still identify high coverage signatures in the remaining code. If the polymorphic code has a relatively small number of variations, `Hancock` can still identify several signatures with moderate coverage that cover most files in the malware family. If all code is polymorphic, with a high degree of variation, `Hancock` will cover very few of the files.

- Finally, the extremely stringent false positive requirement means setting heuristics to very conservative thresholds. Although the heuristics have good discrimination power, they each still eliminate many good signatures. For example, the group count heuristic clusters malware into families based on a single-byte histogram. This splits most malware families into several groups, with large malware families producing a large number of groups. An ideal signature for this family will occur in all of those groups. Thus, for the sake of overall discrimination power, the group count heuristic will reject all such ideal signatures.

4.6.2.1 Sensitivity Study

A heuristic's *discrimination power* is a measure of its effectiveness. A heuristic has good discrimination power if the fraction of false positive signatures that it eliminates is higher than the fraction of true positive signatures it eliminates. These results depend strongly on which other heuristics are in use. We tested heuristics in two scenarios: we measured their *raw discrimination power* when other heuristics were

disabled; and we measured their *marginal discrimination power* when other heuristics were enabled with conservative thresholds.

First, using the August 2008 malware set, we tested the raw discrimination power of each heuristic. Table 4.4 shows the baseline setting, more conservative setting, and discrimination power for each heuristic. The library heuristics (Universal FLIRT, library function reference, and address space) are enabled for the baseline test and disabled to test their own discrimination powers. Using all baseline settings, the run covered 551 malware files with 220 signatures and 84 false positives. Discrimination power is calculated as

$$\frac{\log \frac{\text{FPs}_i}{\text{FPs}_f}}{\log \frac{\text{Coverage}_i}{\text{Coverage}_f}}$$

Table 4.4 shows most of these heuristics to be quite effective. Position deviation and group ratio have excellent discrimination power (DP); the former lowers coverage very little and the latter eliminates almost all false positives. Model probability and code interestingness showed less discrimination power because their baseline settings were already somewhat conservative. Had we disabled these heuristics entirely, the baseline results would have been so overwhelmed with false positives as to be meaningless. All four of these heuristics are very effective.

Increasing the minimum number of malware files a signature must cover eliminates many marginal signatures. The main reason is that, for lower coverage numbers, there are so many more candidate signatures that some bad ones will get through. Raising the minimum coverage can have a bigger impact in combination with diversity-based heuristics, because those heuristics work better with more files to analyze.

Requiring two common signatures eliminated more good signatures than false positive signatures. It actually made the signatures, on average, worse.

Finally, the library heuristics all work fairly well. They each eliminate half to 70% of false positives while reducing coverage less than 30%. In the test for each library

Heuristic	FPs	Coverage	DP
Max pos. deviation (from ∞ to 8,000)	41.7%	96.6%	25
Min file coverage (from 3 to 4)	6.0%	83.3%	15
Group ratio (from 1.0 to .6)	2.4%	74.0%	12
Model log probability (from -80 to -100)	51.2%	73.7%	2.2
Code interestingness (from 13 to 15)	58.3%	78.2%	2.2
Multiple common sig.s (from 1 to 2)	91.7%	70.2%	0.2
Universal FLIRT	33.1%	71.7%	3.3
Library function reference	46.4%	75.7%	2.8
Address space	30.4%	70.8%	3.5

Table 4.4: Raw Discrimination Power

heuristic, the other two library heuristics as well as basic FLIRT functionality were still enabled. This shows that none of these library heuristics are redundant and that these heuristics go significantly beyond what FLIRT can do.

4.6.2.2 Marginal Contribution of Each Technique

Then we tested the effectiveness of each heuristic when other heuristics were set to the Strict thresholds from table 4.1. We tested the tunable heuristics with the 2007-8 malware set with Strict baseline threshold settings from table 4.1. Testing library heuristics was more computationally intensive (requiring that we reprocess the malware set), so we tested them on August 2008 data with baseline Loose threshold settings. Since both sets of baseline settings yield zero FPs, we decreased each heuristic's threshold (or disabled it) to see how many FPs its conservative setting eliminated and how much it reduced malware coverage. Table 4.5 shows the baseline and more liberal settings for each heuristic. Using all baseline settings, the run

Heuristic	FPs	Coverage
Max pos. deviation (from 3,000 to ∞)	10	121%
Min file coverage (from 4 to 3)	2	126%
Group ratio (from 0.35 to 1)	16	162%
Model log probability (from -90 to -80)	1	123%
Code interestingness (from 17 to 13)	2	226%
Multiple common sig.s (from 2 to 1)	0	189%
Universal FLIRT	3	106%
Library function reference	4	108%
Address space	3	109%

Table 4.5: Marginal Discrimination Power

covered 1194 malware files with 206 signatures and 0 false positives.

Table 4.5 shows that almost all of these heuristics are necessary to reduce the FP rate to zero. Among the tunable heuristics, position deviation performs the best, eliminating the second most FPs with the lowest impact on coverage. The group ratio also performs well. Requiring a second common signature does not seem to help at all. The library heuristics perform very well, barely impacting coverage at all. Other heuristics show significantly decreased marginal discrimination power, which captures an important point: if two heuristics eliminate the same FPs, they will show good raw discrimination power, but poor marginal discrimination power.

4.6.3 Single-Component Signature Generation Time

The most time-consuming step in `Hancock`'s string signature generation process is goodware model generation, which, for the model used in the above experiments, took approximately one week and used up all 128 Gbytes of available memory in the

process of its creation. Fortunately, this step only needs to be done once. Because the resulting model is much smaller than the available memory in the testbed machine, using the model to estimate a signature candidate's occurrence probability does not require any disk I/O.

The three high-level steps in `Hancock` at run time are malware pre-processing (including unpacking and disassembly), picking candidate signatures, and applying diversity-based heuristics to arrive at the best ones. Among them, malware pre-processing is the most expensive step, but is also quite amenable to parallelization. The two main operations in malware pre-processing are recursively unpacking malware files and disassembling both packed and unpacked files using IDA Pro. Both use little memory, so we parallelized them to use 15 of the 16 cores. For the 2007-2008 data set, because of the huge number of packed malware files and the decreasing marginal return of analyzing them, `Hancock` disassembled only 5,506 packed files. Pre-processing took 71 hours.

Picking candidate signatures took 145 minutes and 37.4 GB of RAM. 15 minutes and 34.3 GB of RAM went to loading the goodware model. The remainder was for scanning malware files and picking and storing candidate signatures in memory and then on disk.

Generating the final signature set took 420 minutes and 6.07 GB of RAM. Most of this time is spent on running IDA Pro against the byte sequences surrounding the final signatures to output their assembly representation. Without this step, the final signature generation step should have taken only a few minutes.

4.6.4 Multi-Component Signatures

We tested MCS signatures with 2 to 6 components, with each part being 16 bytes long. We used a 3.0 GB goodware set to select component candidates and tested

number of components	Permitted component FPs	Coverage	# Signatures	# FPs
2	1	28.9%	76	7
2	0	23.3%	52	2
3	1	26.9%	62	1
3	0	24.2%	44	0
4	1	26.2%	54	0
4	0	18.1%	43	0
5	1	26.2%	54	0
5	0	17.9%	43	0
6	1	25.9%	51	0
6	0	17.6%	41	0

Table 4.6: Multi-Component Signature results

for false positives with a 34.9 GB set of separate goodware.[2] Table 4.6 shows the coverage and false positive rates when 0 or 1 components could be found in the smaller goodware set.

We first observe that permitting a single component of an MCS to be an FP in our small goodware set consistently results in higher coverage. However, from 2- and 3-component signatures, we also see that allowing a single component FP results in more entire MCS FPs, where all signature components occur in a single goodware file.

We can trade off coverage and FP rate by varying the number of signatures components and permitted component FPs. Three to five part signatures with 0 or 1 allowed FPs seems to provide the best tradeoff between coverage and FPs.

Since we applied so few heuristics to get these results, beyond requiring the existence of the multiple, disjoint signature components which make up the signature, it is perhaps surprising that we have so few MCS FPs. We explain this by observing that although we do not limit MCS components to code bytes, we do apply all the library code reducing heuristics through IDA disassembly described in Section 4.3.2.

Also, the way in which signature components are selected from contiguous runs

[2]This final goodware set was smaller than in SCS tests because of the difficulty of identifying shorter, 16-byte sequences.

of identical bytes may reduce the likelihood of FPs. If a long, identical byte sequence exists in a set of files, the 16 byte signature component with lowest probability will be selected. Moreover, no other signature component will be selected from the same run of identical bytes. Thus, if malware shares an identical uncommon library (which we fail to identify as a library) linked in contiguously in the executable, at most one signature component will be extracted from this sequence of identical bytes. The other components must come from some other shared code or data.

Finding candidate signatures took 1,278 minutes and 117 GB of RAM. Picking the final signature sets took between 5 and 17 minutes and used 9.0 GB of RAM.

4.6.5 Comparison of Multi-Component Signatures with Single Component Signatures

Comparing our best coverage with no FPs for MCS to single-component signatures using our best combination of heuristics, we see that we get approximately double the coverage with MCS signatures. Moreover, unlike single signatures, we used few heuristics to get these results, beyond requiring the existence of the multiple signature components which make up the signature. Future work could include applying the heuristics developed for single component signatures to MCS, with the understanding that some heuristics (like the interestingness heuristic) will be difficult to apply on a sequence of only 16 bytes.

The MCS with more than 3 parts require more memory to store than the corresponding single component signatures. Depending on the scanning architecture, it may also be slower to scan for MCS signatures than single component signatures.

The final FP check uses a smaller goodware set than the one used for SCS because we had to build a more finely indexed data structure to support queries for the shorter, 16-byte sequences. Future work should include re-indexing the larger SCS goodware set.

More manual analysis of single component signatures was performed, and the heuristics were tightened beyond the point where the run had zero FPs, until the signatures looked good by manual analysis. A similar, in-depth analysis of MCS was not performed.

4.7 Conclusion

Given a set of malware files, an ideal string signature generation system should be able to automatically generate signatures in such a way that the number of signatures required to cover the malware set is minimal and the probability of these signatures appearing in goodware programs is also minimal. The main technical challenge of building such string signature generation systems is how to determine how FP-prone a byte sequence is without having access to even a sizeable portion of the world's goodware set. This false positive problem is particularly challenging because the goodware set is constantly growing, and is potentially unbounded. In the `Hancock` project, we have developed a series of signature selection and filtering techniques that collectively could remove most if not all FP-prone signature candidates while maintaining a reasonable coverage of the input malware set. In summary, the `Hancock` project has made the following research contributions in the area of malware signature generation:

- A scalable goodware modeling technique that prunes away unimportant nodes according to their relative information gain and merges submodels without losing information so as to scale to very large training goodware sets,
- A set of diversity-based signature filtering techniques that eliminate signature candidates when the set of malware programs they cover exhibit high diversity, and
- The first known string signature generation system that is capable of creating

multi-component string signatures which have been shown to be more effective than single-component string signatures.

Although `Hancock` represents the state of the art in string signature generation technology, there is still room for further improvement. The overall coverage of `Hancock` is lower than what we expected when we started the project. How to improve `Hancock`'s coverage without increasing the FP rate of its signatures is worth further research. Although the multi-component signatures that `Hancock` generates are more effective than single-component signatures, their actual run-time performance impact is unclear and requires more thorough investigation. Moreover, there could be other forms of multi-component signatures that `Hancock` does not explore and therefore deserve additional research efforts.

CHAPTER V

DUET: Integrating Dynamic and Static Analysis for Malware Clustering

5.1 Introduction

The growing popularity of automatic malware-creation toolkits, which allow even marginally skilled attackers to create and customize malware programs, has resulted in a plethora of malware variants. Clustering, a valuable tool in malware analysis, partitions new malware programs into similar groups (clusters); samples grouped together are similar, whereas those in different groups are dissimilar. Often, in a malware analysis, little prior knowledge exists for new malware samples, and hence, clustering plays a vital role in initially processing new incoming programs. For instance, the automatic, efficient clustering of malware samples into groups will allow analysts to prioritize, allocating precious human resources for more important, distinct malware programs. Second, it will enable automatic labeling of new incoming samples based on their association with existing clusters. Finally, clustering makes it easier to generalize previous signatures, remedy procedures and mitigation techniques for new variants.

There are many clustering techniques one can use to analyze malware samples. Different clustering algorithms or parameter settings are likely to generate distinct,

or even conflicting, clustering results. Even multiple runs of the same algorithm, with identical parameter settings, may produce different results due to the random initialization or stochastic learning process [118]. Choosing "the best" algorithm or settings therefore represents a challenging task because different approaches have their respective advantages and limitations, which often depend on properties such as data distribution, pre-processing procedures, anti-analysis techniques used by malware programs, etc. Moreover, it is not uncommon for different clustering algorithms to generate inconsistent, or even contradictory results, when applied to the same dataset, a phenomenon originating from their individual biases and strengths towards particular sets of data. Since no single algorithm performs optimally across all various data sets, a broad range of clustering algorithms have been proposed to tackle the malware clustering problem [10, 13, 64, 81, 86, 88, 113], each of which has its own merits and demerits. As discussed in Chapter III, there are two fundamental approaches to malware clustering, i.e., based on *static features* or *dynamic behavior*. The static approach is efficient and capable of covering all code paths, including parts of the programs that are usually not executed. However, its performance suffers from low-level mutation techniques, such as packing and obfuscation. In contrast, dynamic analysis encounters the exact opposite situation. While it achieves great resilience to low-level obfuscation making it potentially well-suited for generalizing unknown malware variants, it often performs poorly when handling trigger-based malware programs. Consequently, malware samples that can be effectively analyzed by these two approaches are usually different, making it challenging to choose a single, optimal clustering algorithm. By combining these two approaches, we can exploit their respective strengths while diminishing their weaknesses.

In this chapter, instead of focusing on the development of a single clustering algorithm that only works for a narrow range of datasets, we design a unified clustering framework, effectively combining results from different clustering algorithms

based on the concept of *cluster ensemble*. More specifically, given a set of clusterings $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_m$, cluster ensemble attempts to derive a single clustering $\mathcal{C}$ that, according to certain criteria, is in as much agreement as possible with the original m clusterings. Cluster aggregation overcomes multiple shortcomings of any single clustering algorithm, which cannot possibly cover all malware samples alone, often resulting in better quality clustering with greater coverage. For instance, the dynamic approach cannot handle malware that denies execution if run in a virtual environments, such coverage gaps can often be resolved by utilizing the complementary static approach. In addition, cluster ensemble exploits the consensus among different clustering algorithms to derive new clusters, reinforcing the grouping's confidence level and improving the clustering quality. One caveat worth consideration is the conflicting results generated by different clustering algorithms. For example, dynamic approaches may mistakenly cluster together all malware programs capable of detecting virtual environments and halting its execution, as they all invoke a smaller number of similar API calls. In contrast, static clustering can probably separate such malware into distinct groups based on other features. While static clustering may err by grouping together programs packed with the same customized packers, these samples are likely to be handled properly and separated by dynamic clustering. However, if static and dynamic approaches carry equal weights when reconciling their results, the ensemble algorithm can, at best, make random choices. In this chapter, we address this problem through *Clustering Quality Measurement*, which measures how strong or conclusive the data points are within each group based on criteria such as scatter, density and the number of data points in a cluster. By assigning each cluster a quality score, high-quality clusters carry more weight when reconciling the conflicting results, thus improving the clustering algorithm.

In this chapter, we investigate and compare the effectiveness of various cluster-ensemble methods, including hyper-graph partitioning [95], agglomerative algorithm

[38], etc. The rest of this chapter is organized as follows. Section 5.2 gives a brief overview of DUET. Section 5.3 presents the malware trace collection system and the clustering algorithm. Sections 5.4 and 5.5 detail the proposed cluster ensemble algorithms. Section 5.6 presents the comprehensive evaluation of the DUET system, and Section 5.8 concludes the chapter.

5.2 System Overview

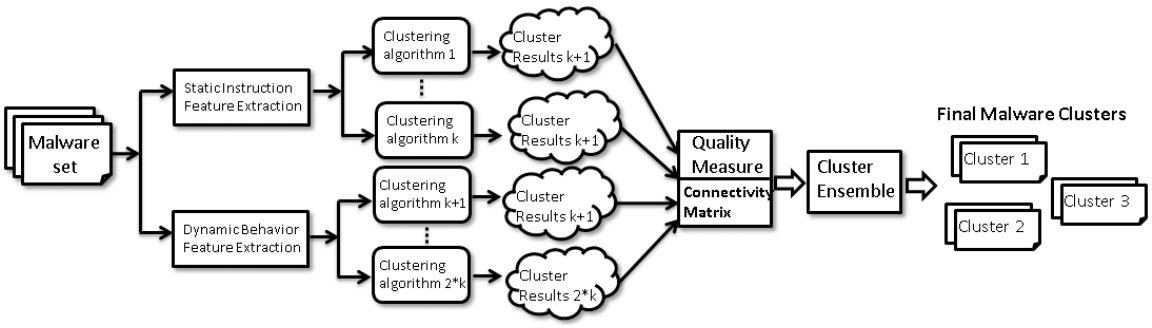


Figure 5.1: An overview of DUET

Here we briefly describe the system architecture of DUET, which is illustrated in Figure 5.1. Given a set of malware programs, DUET first uses two different feature extractors to derive (1) instruction-based features, i.e., opcode sequences, and (2) dynamic behavior features, i.e., system call traces. These features are then transformed into a vector for the purpose of clustering. Second, base clustering results are generated by applying (different) clustering algorithms with different parameters on the dataset. To improve scalability, the same prototype-based clustering algorithm and hashing tricks as in MutantX (Chapter III) are also used in dynamic behavioral clustering. Third, a connectivity matrix is constructed for each individual clustering algorithm to represent the clustering results based on the quality measures. All matrices are combined to form a master matrix, where various ensemble methods can be applied in order to derive the final results.

5.3 Malware Clustering Using Run-time Traces

This section describes the system we developed to perform dynamic analysis on malicious samples and extract their run-time behavior in terms of system calls. Because most malware samples target the Windows operating system, we use the BindView's STrace [16] (the Windows version of Linux STrace utility) to intercept and record all system calls invoked by the malware program, as well as their detailed arguments. STrace consists of two components: a device driver, which patches the kernel's system call table to collect all call traces, and a user-space application, which loads and communicates with the driver it to retrieve traces. All the malware samples are executed in a VMware virtual machine running the Windows XP system. We make use of VMware VIX [109] API to automate and parallelize the running of malware programs. Each malware program is monitored for 2 minutes, and the resulting system call traces are transferred from the virtual machine to the host machine. After transferring the collected traces, the virtual machine is reset to the clean-state snapshot, preventing interference between malware programs.

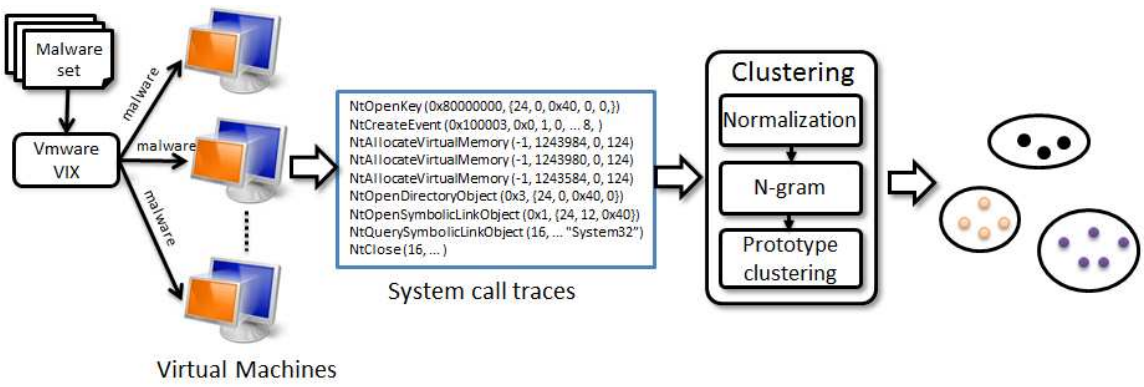


Figure 5.2: Malware clustering based on dynamic behavior

STrace produces a textual format of system call traces (Figure 5.2) which is difficult to use for the automatic analysis of malware behavior. To address this difficulty,

we employ the idea proposed in [88] and encode each system call with a tuple '(category, operation)' shown in Table 5.1, where a category represents a group of system calls that operate on a similar type of objects (e.g., registry, file systems, DLLs or processes), and an operation specifies a particular call function. For example, in Table 5.1, the tuple '(0x03, 0x05)' indicates that the system call belongs to the "File System" category and is performing the "DeleteFile" operations. Furthermore, we assign the same operation value to those system calls that can achieve identical results, such as OpenProcess and NtOpenProcess, RegOpenKey and RegOpenKeyEx. For instance, 4 system calls that can be used to create a new process are encoded with the same tuple, '(0x0A, 0x02)'. This canonicalization of function variations enables a more generalized representation of system call traces and improves the clustering accuracy.

Because typical malware behavior patterns, such as modifying the registry keys and file systems, can be reflected in the system call sequences, we apply the standard n-gram analysis as in `MutantX`, embedding the encoded system call sequences into a fixed-length feature vector whose distance represents the similarity between malware behaviors. However, unlike in `MutantX`, where the feature vector elements represent how often a specific n-gram of *instructions* occurs, dynamic behavioral clustering uses "binary features" (i.e., 0 or 1) where each feature vector element represents the absence or presence of a specific n-gram of *system calls* in the call traces. Using binary features reduces the influence of external factors, such as the length of traces, the redundancy of the behavior and the alphabet of the n-gram. In practice, depending on the monitoring period and system condition, the number of system calls in a call trace can vary significantly, even for identical malware programs. In addition, when executed in a loop, certain system calls may be repeated thousands of times, considerably skewing the values in the feature vector. Consequently, this may introduce an implicit bias and render the comparison of samples with small and large traces

	Category	Operation	System Call
File System	0x03	0x01	CreateFile
	0x03	0x02	CopyFile
	0x03	0x02	CopyFileEx
	0x03	0x03	ReadFile
	0x03	0x03	ReadFileEx
	0x03	0x04	WriteFile
	0x03	0x04	WriteFileEx
	0x03	0x05	DeleteFile
DLL Handling	0x02	0x01	LoadLibrary
	0x02	0x01	LoadLibraryEx
	0x02	0x02	GetProcAddress
	0x02	0x03	GetModuleHandle
	0x02	0x04	GetModuleFileName
Registry Handling	0x09	0x01	RegOpenKey
	0x09	0x01	RegOpenKeyEx
	0x09	0x02	RegCreateKey
	0x09	0x02	RegCreateKeyEx
	0x09	0x03	RegEnumKey
Process Handling	0x0A	0x01	OpenProcess
	0x0A	0x01	NtOpenProcess
	0x0A	0x02	CreateProcess
	0x0A	0x02	CreateProcessInternal
	0x0A	0x02	CreateProcessAsUserr
	0x0A	0x02	NtCreateProcess

Table 5.1: Encoding of sample system calls

inaccurate. To compensate for this bias, the value of each component in the feature vector $v(x)$ is limited to be either 0 (n-gram feature is absent) or 1 (n-gram feature is present). Additionally, to minimize the impact of inconsistent trace length, the feature vector is normalized, setting its length equal to 1 (i.e., $||v(x)|| = 1$) and ensuring that the difference between vectors depends only on the presence/absence of certain behavioral patterns. After feature extraction and vector encoding, the similarity of behavior patterns between different malware programs can be expressed as geometrical distance $d(x, y)$ in the vector space:

$$d(x, y) = ||v(x) - v(y)|| = \sqrt{\sum_{i=1}^{n}(v_i(x) - v_i(y))^2}.$$

Note that, due to normalization, the values of d range from 0 for identical behavior, to $\sqrt{2}$ for completely different behavior. Using this distance definition for assessing behavioral similarity, the standard prototype-based clustering algorithm is applied to malware samples, creating a particular cluster for input into the cluster ensemble algorithm.

5.4 Cluster Ensemble

The motivation for using cluster ensemble is to combine the strengths of different clustering algorithms and to improve the quality and the robustness of clustering results. Cluster ensemble exploits the diversity of different clustering algorithms, reconciling their discrepancies on a data set, to produce a result that performs as well as, if not better than, the results of any single clustering algorithm alone. In this section, we first demonstrate the limitations of dynamic and static analyses using real-world malware data, illustrating the potential for further improvement with the cluster ensemble. Next, we discuss potential cluster-ensemble methods that can be used to integrate static and dynamic analysis results.

5.4.1 Motivating Examples

For certain malware, the respective limitations of dynamic and static approaches can render them ineffective when used alone, producing few dynamic API traces or static instruction features. To verify this, we run static and dynamic analyses on a set of 5,647 real-world malware samples (detailed categorization of these samples will be presented in Section 5.6) to collect their code instruction and dynamic system call traces, extracting n-gram features. Because extraction of n-gram features requires the collected sequences to have at least n system calls or instructions, we first measure the number of malware samples that satisfy this constraint and whose feature vector (from either dynamic API sequences or static instruction sequences) can be successfully extracted. By applying dynamic and static n-gram analyses, we gain insight into the capabilities and shortcomings of both approaches, revealing their complementary nature. Table 5.2 lists the feature extraction results from these approaches. The table clearly shows that any single approache alone is unable to analyze all the samples and, in general, fails to extract features from 12–17% of all malware binaries. For instance, 645 samples do not run in our virtual machine, producing no system calls; furthermore, around 25 malware programs only make a single system call, (i.e., TerminateProcess), presumably due to detection of the virtual machine environment and then immediate termination. On the other hand, IDA fails to extract any static instructions from 655 binaries due to the underlying obfuscation and packing techniques. However, a combination of dynamic and static approaches yields much better results: the percentage of malware samples that can be successfully analyzed improves to 98.72%, and only 72 malware samples are able to evade both approaches. Although the final coverage depends on the cluster-ensemble algorithm, this preliminary experiment shows that aggregating different analysis approaches has the potential to achieve more robust and complete clustering.

	# of successfully processed malware	percentage	# of failed malware	percentage
Behavior 3 gram	4971	88.03	676	11.97
Behavior 4 gram	4943	87.53	704	12.47
Static 3 gram	4799	84.98	848	15.02
Static 4 gram	4703	83.28	944	16.72
Behavior+Static	5575	98.72	72	1.28

Table 5.2: Number of malware samples whose features can be extracted by static, dynamic, and both approaches. The total number of malware samples is 5647

5.4.2 Problem Formulation

Consider a set of n malware programs, $X = x_1, x_2, \ldots, x_n$, and a set of T clusterings of X, $\mathcal{C} = \{\mathcal{C}^1, \mathcal{C}^2, \ldots, \mathcal{C}^T\}$. Each clustering, $\mathcal{C}^t$, $t = 1, 2, \ldots, T$, is a partition of X into k disjoint clusters, i.e., $\mathcal{C}^t = \{C_1^t, C_2^t, \ldots, C_k^t\}$ where $\bigcup_{i=1}^{k} C_i^t = X$ and $C_i^t \cap C_j^t = \phi$, $\forall i \neq j$. Let $L^t(x)$ denote the label of the cluster to which the malware program, x, belongs, i.e., $L^t(x) = j$ if and only if $x \in C_j^t$. Also, k could be different for different clusterings. With these T clusterings, the cluster ensemble is defined as a consensus function Γ [96] that maps a set of clusters to an integrated clustering:

$$\Gamma : \{\mathcal{C}^t | t \in \{1, 2, \ldots, T\}\} \rightarrow C$$

if the relative importance of each individual clustering is not known *a priori*, a natural goal of cluster ensemble is to find the final clustering, $\mathcal{C}$, that shares the most commonality with the constituent clusterings [38].[1] To measure the similarity or dissimilarity between clusterings, we define a connectivity matrix, $M(\mathcal{C}^t)$, for each clustering $\mathcal{C}^t$. The connectivity matrix is an $n \times n$ pair-wise similarity matrix defined for all malware programs and it represents the structural information of a particular clustering. Since the matrix size is fixed, it provides a method for aligning differ-

[1]The algorithm can be easily generalized to the case where some clusterings may carry more weights than others

ent clusterings onto the same space even when the number of clusters varies across different approaches. Specifically, $M(\mathcal{C}^t)$ is defined as in [118]:

$$M_{ij}(\mathcal{C}^t) = \begin{cases} 1 & \text{if sample } x_i \text{ and } x_j \text{ belong to the same cluster in } \mathcal{C}^t \\ 0 & \text{Otherwise} \end{cases}$$

Then, the difference between two clusterings, $\mathcal{C}^a$ and $\mathcal{C}^b$, in determining whether x_i and x_j are in the same cluster can be expressed as:

$$d_{i,j}(\mathcal{C}^a, \mathcal{C}^b) = |M_{ij}(\mathcal{C}^a) - M_{ij}(\mathcal{C}^b)|.$$

From this, we can define the distance between two clusterings, C^a and C^b, as the number of malware pairs for which the two clusterings disagree [38]:

$$d(\mathcal{C}^a, \mathcal{C}^b) = \sum_{i,j=1}^{n} d_{i,j}(\mathcal{C}^a, \mathcal{C}^b) = \sum_{i,j=1}^{n} |M_{ij}(\mathcal{C}^a) - M_{ij}(\mathcal{C}^b)| = \sum_{i,j=1}^{n} (M_{ij}(\mathcal{C}^a) - M_{ij}(\mathcal{C}^b))^2.$$

Cluster ensemble strives to find a consensus clustering, $\hat{\mathcal{C}}$, that is closest to all of the given clusterings, i.e., that minimizes the average distance between $\hat{\mathcal{C}}$ and $\{\mathcal{C}^t | t \in \{1, 2, \ldots, T\}$:

$$\mathcal{C}^{opt} = \arg\min_{\hat{\mathcal{C}}} \sum_{t=1}^{T} d(\mathcal{C}^t, \hat{\mathcal{C}}) \tag{5.1}$$

$$= \arg\min_{\hat{\mathcal{C}}} \sum_{t=1}^{T} \sum_{i,j=1}^{n} (M_{ij}(\mathcal{C}^t) - M_{ij}(\hat{\mathcal{C}}))^2. \tag{5.2}$$

Since $\sum_{t=1}^{T} d(\mathcal{C}^t, \hat{\mathcal{C}})$ is a convex function, minimizing it makes the optimal connectivity matrix $M_{i,j}^{opt}(\hat{\mathcal{C}})$ become the average connectivity matrix [118]:

$$M_{i,j}^{opt}(\hat{\mathcal{C}}) = \frac{1}{T}\sum_{t=1}^{T} M_{ij}(\mathcal{C}^t),$$

where M^{opt} represents the integrated connectivity relationship between data samples from different clusterings. When weights are not assigned to the different clusterings, the values of $M_{i,j}^{opt}$ indicate the average number of times that x_i and x_j are clustered together; they range between 0 and 1, where 1 means all the clusterings agree that x_i and x_j belong to the same cluster, and 0 means none of the clusterings groups x_i and x_j together. Given $M_{i,j}^{opt}$, our goal is to derive $\mathcal{C}^{opt} = \{C_1^{opt}, C_2^{opt}, \ldots, C_k^{opt}\}$ from $M_{i,j}^{opt}$.

5.4.3 Clustering Based on Ensemble Distance Matrix

Several methods have been proposed to generate a final clustering from the optimal connectivity matrix $M_{i,j}^{opt}$ [38, 95, 118]. Here we employ the following algorithms.

Simple threshold algorithm: A simple strategy to generate the final clustering is to use a single threshold. For each sample pair (x_i, x_j), if $M_{i,j}^{opt}$ is greater than the threshold, x_i and x_j are assigned to the same cluster. If the samples are previously assigned to different clusters, these clusters are merged. When the threshold is set to 0.5, the algorithm becomes a majority vote, and two samples are clustered into the same group only if at least half of the input clusterings agree. Finally, all remaining unclustered samples each form a single-element cluster.

The balls algorithm [38]: The basic idea of this algorithm is to find a set of samples that are close to each other (within a ball) and far from others. After finding such a cluster, the constituent samples are removed from the data set, and the clustering continues with the remaining samples. Because the problem of finding the globally optimal clusters is NP-complete, a greedy algorithm can be used to create a bounded approximation to the optimal solution. Viewing the connectivity matrix, M^{opt}, as a graph's adjacency matrix, where each $M_{i,j}^{opt}$ represents the edge's

weight connecting x_i and x_j, the algorithm sorts the samples in decreasing order of their edges' total weights. At each step, the algorithm chooses the first unclustered sample, x_u, and finds a set of samples, $V = x_{v1}, x_{v1}, \ldots, x_{vk}$ whose connectivity to x_u is greater than the threshold, β. Then, their union $V \cup x_u$ forms a cluster.

The agglomerative algorithm[38]: This is a standard bottom-up clustering approach. It starts by placing all samples as singleton clusters. Next, it recursively merges the two clusters with the smallest distance; repeating until the distance between any pair of existing clusters is larger than a threshold, h. In DUET, we define the distance between clusters as $1 - \hat{c}$ where $\hat{c}$ is the average connectivity between every pair of samples from two clusters. If the threshold, h, is set to $1/2$, the agglomerative algorithm is guaranteed to create clusters where the average connectivity of any pair of nodes is at least $1/2$ (i.e., at least half of the original clusterings are in agreement).

Hypergraph partition algorithm [95]: Essentially, the cluster ensemble repartitions the original dataset based on other clusterings' indications of strong connections. Therefore, it can also be formulated as a hypergraph partition problem, where each sample is a vertex in the hypergraph, and the hyperedge between vertices is weighted based on M^{opt}. In this case, the goal is to cut the minimum set of edges such that the remaining subgraphs consist of connected components corresponding to new clusters. Hypergraph partitioning is a well-studied area and many existing algorithms can be applied to efficiently find the minimum cut. As suggested in [47], we choose the hMETIS [63] package because of its good performance and scalability.

5.5 Improving Cluster Ensemble with Cluster-Quality Measure

In effect, standard cluster ensemble approaches weight all clusters equally. For instance, M_{ij} is set to 1 if two data points, x_i and x_j, are in the same cluster, regardless of the cluster quality. However, since clustering is essentially unsupervised learning without prior knowledge of the underlying data distribution, every clustering algorithm implicitly or explicitly assumes some data model and may produce erroneous or meaningless clusters when these assumptions are not satisfied by the sample data. In other words, because of their exploratory nature, most clustering algorithms will create clusters even for data points that have little or no correlation, leading to false clusters and negatively impacting the final results of the integrated clusters.

To overcome this limitation, we propose to improve the cluster ensemble algorithm by differentiating clusters that have non-random structures from those that are created artificially by the clustering algorithms, weighting high-quality clusters more in the final results. This is particularly important when reconciling conflicting clusters because different clustering algorithms(e.g., static feature and dynamic behavior based approaches), handle different types of malware programs with varying levels of effectiveness. Ideally, we should give more weight to clusterings that group particular samples well. For instance, we want to rely more on dynamic analysis when clustering malware programs that are heavily obfuscated. Unfortunately, such information is not readily available, so we use cluster-quality measurements as an indirect way to allow the cluster ensemble method to bias towards high-quality clusters. High-quality clusters are compact and separated well from other clusters, indicating that the data points likely share a strong bond. Such clusters should have a greater influence on the final ensemble results, increasing the probability of preserving the connections between samples. In contrast, lower-quality clusters, where there is less correlation

and more diversity among samples, should hold less sway over the final ensemble, avoiding potential mis-grouping of unrelated data points.

Hence, our objective is to (1) define quality metrics that evaluate the "goodness" of clustering by looking at the intra- and inter-cluster correlations between data points and (2) incorporate the metrics in the ensemble algorithms. More specifically, we define the following measures of cluster quality:

- **Cluster Cohesion (C_o)**, also called compactness or tightness, determines how closely objects in a cluster are related. Formally, a cluster's cohesion is defined as the average link weight of its connectivity graph of the cluster. For a cluster, C_i^t, created by a specific clustering algorithm, t, cohesion can be calculated using the *proximity* function between two input samples:

$$C_o(C_i^t) = \frac{2}{|C_i^t| * (|C_i^t| - 1)} \sum_{x \in C_i^t; y \in C_i^t; x < y} proximity(x, y).$$

  The proximity function is usually the distance function (e.g., normalized Euclidean distance) specific to the clustering algorithm. Notice that the according to this definition, a smaller C_o implies a more cohesive cluster.

- **Cluster Separation (C_s)** measures how well or distinct a cluster is separated from other clusters. The separation between two clusters, C_i^t and C_j^t, can also be defined with the proximity function:

$$C_s(C_i^t, C_j^t) = \frac{1}{|C_i^t| * |C_j^t|} \sum_{x \in C_i^t; y \in C_j^t} proximity(x, y).$$

  Then, the average separation, $\bar{C_s}(C_i^t)$, of the cluster C_i^t from all other clusters

can be computed as:

$$\bar{C}_s(C_i^t) = \frac{1}{k-1} \sum_{j=1;j\neq i}^{k} \sum_{x\in C_i^t; y\in C_j^t} proximity(x,y)$$

where k is the number of clusters.

Intuitively, high-quality clusters (i.e., those with small cohesion and high separation), should more likely be in the final ensemble results. `DUET` exploits such "goodness" information and incorporates it into the ensemble process. Well-formed clusters with strong bonds are likely to be preserved in the final results, while clusters with no natural linkage are discouraged from influencing the ensemble process. To achieve this goal, we use a weighted boost score β:

$$\beta = \omega_o(1 - C_o) + \omega_s C_s$$

where $\omega_o + \omega_s = 1$. Since C_o and C_s both lie between 0 and 1, the boost score also ranges from 0 and 1, with a higher value indicating a better cluster. We use β to augment the connectivity matrix of each member clustering and increase M_{ij} if x_i and x_j are in a well-formed cluster. More specifically, the new boosted connectivity matrix is computed as:

$$M_{i,j}^B(\mathcal{C}^t) = M_{i,j}(\mathcal{C}^t) \times (1 + \beta_{L^t(i)}^t)$$

and the cluster ensemble algorithm described in Section 5.4 is applied on the boosted connectivity matrix to derive the final clustering results.

5.6 Evaluation

In this section, we conduct a set of experiments using a large number of real-world malware samples to evaluate the cluster ensemble methods discussed thus far. First, we present the experimental results of dynamic-behavior-based clustering algorithms and quantitatively compare them with static-feature-based approaches, confirming our conjecture that the two techniques cover different sets of malware samples, and therefore, cluster ensemble has the potential to improve the overall quality of clustering. In the second part of the experiments, we assess the proposed cluster ensemble algorithms—including single-threshold, balls, agglomerative and hypergraph-partitioning algorithms—and demonstrate their improvements over the single clustering algorithm in terms of the precision and sample coverage. Finally, we show that the ensemble results can be further improved by taking into consideration the quality of each member cluster.

5.6.1 Malware data set

To evaluate `DUET`, we use the same malware dataset for both static and dynamic clustering algorithms. The dataset contains 5,647 malware files with known class labels which were collected and analyzed by Symantec malware analysts. The number of samples in each family is summarized in Table 5.3. The malware binaries are executed in the virtual machine, and behavior traces are collected and analyzed by `MutantX` to extract feature vectors.

In order to ensure the meaningful representation of malware's semantics, the clustering algorithm imposes a threshold constraint, discarding the malware programs with less than 10 n-grams. This is to prevent the use of overly generic features in malware clustering. Further, when malware has too few extracted n-grams, it is likely that the extraction process—either disassembling or dynamic monitoring—encountered certain problems, making it unsafe to include them in clustering. As

Family	#	Family	#	Family	#
Pilleuz	500	Bredolab	362	Mabezat	129
Virut	500	Vundo	334	Qakbot	44
Silly	500	Almanahe	327	Waledac	41
Fakeav	500	Tidserv	242	Ackantta	36
Koobface	496	Sasfis	219	Mebroot	26
Banker	489	Gammima	206	Hotbar	21
Zbot	486	Graybird	189		

Table 5.3: Malware families of the reference data set

a result, some malware samples in Table 5.2 cannot be successfully clustered. We summarize the number of malware programs that pass our ten n-gram constraint for different clustering algorithms in Table 5.4, expressing the maximum malware coverage achievable by each individual clustering. Table 5.4 once more demonstrates the potential benefits of ensemble methods: the combination of static and behavioral approaches significantly increases the number of analyzable malware programs, raising the percentage from less than 80% to almost 97%.

	# of malware with more than 10 n-grams	percentage	# of malware with less than 10 n-grams	percentage
Behavior 3 gram	4026	71.29	1621	28.71
Behavior 4 gram	4038	71.51	1609	28.49
static 3 gram	4622	81.85	1025	18.15
static 4 gram	4605	81.55	1042	18.45
behavior+static	5454	96.58	193	3.41

Table 5.4: Number of malware samples with more than 10 n-grams and the total number of malware samples is 5647

5.6.2 Behavioral clustering results

Next, we present the evaluation results of the proposed behavioral clustering component in `DUET`. The system executes each malware program for 120 seconds, collecting system call traces and converting them into a feature vector. The prototype-based clustering algorithm is then applied on the feature vectors and creates a set of clusters

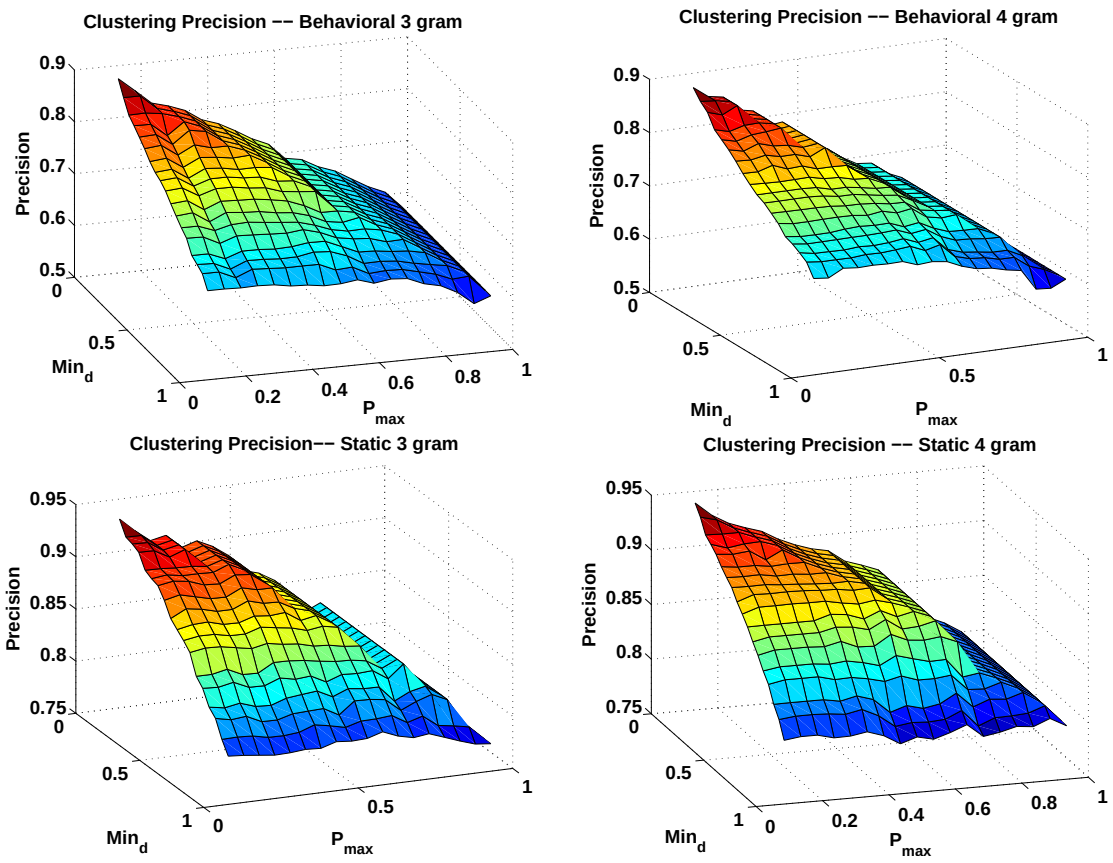


Figure 5.3: Clustering precision

$C = C_1, C_2, \ldots, C_c$. The performance of the clustering algorithm is measured using two metrics: *precision* and *coverage*. Precision is used to assess the accuracy of behavioral clustering in terms of how well the individual clusters agree with the original malware classes. More formally, if we assume the malware samples are grouped into a set of clusters, $O = O_1, O_2, \ldots O_o$, according to their family labels, then the precision, P, is defined as:

$$P = \frac{1}{n} \sum_{i=1} cmax(|C_i \cap O_1|, |C_i \cap O_2|, \ldots, |C_i \cap O_o|).$$

The second metric, coverage, measures the percentage of malware programs that can be successfully clustered after excluding single-member clusters, which are inevitably created for sample outliers. For example, the agglomerative algorithm be-

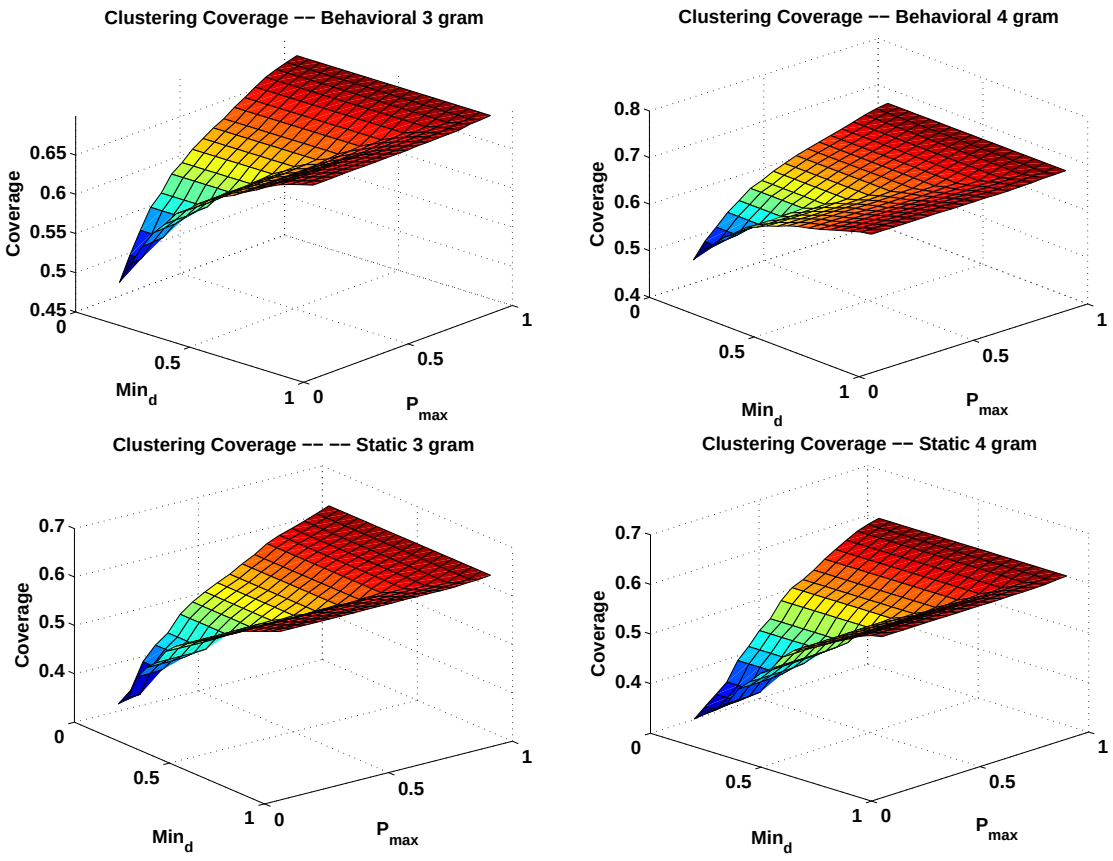


Figure 5.4: Clustering coverage

gins by making each sample an individual cluster; it then recursively merges nearby clusters until a certain criterion is reached, leaving outliers in their own, singular clusters. Since it indicates how well clustering algorithms produce useful clusters, we also consider coverage an important metric.

We evaluate the performance of `DUET`'s dynamic analysis component , as we did for `MutantX`. We vary three parameters—P_{max}, Min_d and the number of grams—required by the prototype-based clustering algorithm and plot the clustering results in Figures 5.3 and 5.4. For comparison, we also plot static-feature-based clustering results in the same graph. From these figures, we observe that with proper parameter selection, all the clustering algorithms are able to cluster malware samples with precision ranging from 70 to 90%. Additionally, static-feature-based clustering, with precision approaching 95%, generally outperforms behavior-based clustering. However, none of

these systems attains perfect coverage and most of them can only create good clusters for only 50–70% of samples. Examining both figures, we can see that there is a natural trade-off between precision and coverage; an increase in precision is accompanied by a decrease in coverage and vice versa. The reason for this phenomenon is that in order to achieve a higher precision, clustering algorithms must avoid merging unrelated malware programs into the same cluster. While this produces high-quality clusters for some samples, many more remain uncovered in singular clusters. For example, as Min_d and P_{max} both decrease toward 0.1, precision is improved; a small P_{max} dictates that each prototype sample includes only extremely close data points within its range, while a small Min_d terminates the merging process early to avoid combining unrelated classes. Together, this combination ensures that only incredibly similar samples are clustered together, excluding a large portion of samples with moderate resemblance to each other. Thus, it leads to a rapid drop in coverage from 75 to 45% for the dynamic approach and below 30% for the static approach.

5.6.3 Evaluation of Cluster Ensemble

In this subsection, we present the evaluation results of the effectiveness of malware clustering using the proposed cluster ensemble methods. The goal is to show how cluster ensemble can be used to improve the quality and robustness of existing clustering algorithms. Using the data set described in Section 5.6.1, we evaluate cluster ensemble for two scenarios. In each scenario, cluster ensemble examines 8 input clustering—two clusterings with different P_{max} and Min_d values, for each approach combination of 3-gram/4-gram and static/behavioral. In the first scenario, the *best-case* scenario, P_{max} and Min_d are selected to give each approach combination two clusterings: one optimized for precision (thus with low coverage) and one optimized for coverage (thus with low precision). Notice that this scenario represents an *ideal*, not a *realistic*, setting. In practice, we would not be able to determine which values

of P_{max} and Min_d lead to the highest precision or coverage. As a result, the best-case scenario simply helps us evaluate the optimal performance of cluster ensemble under ideal conditions and compare it to the best performance achieved by individual clusterings. Our second scenario, the *random* scenario, offers a more realistic evaluation by randomly choosing P_{max} and Min_d for the 8 input clusters. Tables 5.6.3 and 5.6 list the parametric values of the individual input clusterings in these two scenarios. From the tables, we can see that the precision for the best-case scenario's clusterings ranges from 0.69 to 0.88, while the random scenario's range of 0.57 to 0.83 is slightly lower. The range of coverage is usually 50–68% for behavioral clusterings and 54–74% for static clusterings, which are about 10% lower than the maximum coverage shown in Table 5.4. Next, we will show that cluster ensemble is to leverage the different perspectives of each input clustering to improve malware coverage and precision.

	Min_d	P_{max}	precision	coverage
behavior 3 gram (Best precision)	0.15	0.10	0.87	51.5%
behavior 3 gram (Best coverage)	0.85	0.45	0.69	67.9%
behavior 4 gram (Best precision)	0.10	0.15	0.88	51.1%
behavior 4 gram (Best coverage)	0.40	0.95	0.68	68.1%
static 3 gram (Best precision)	0.65	0.20	0.86	57.4%
static 3 gram (Best coverage)	1.30	0.30	0.70	74.4%
static 4 gram (Best precision)	0.30	0.70	0.85	56.7%
static 4 gram (Best coverage)	1.30	0.75	0.70	71.0%

Table 5.5: Parametric settings for the best scenario

	Min_d	P_{max}	precision	coverage
behavior 3 gram	0.20	0.60	0.71	64.7%
behavior 3 gram	0.30	0.20	0.80	57.9%
behavior 4 gram	0.50	0.20	0.77	62.2%
behavior 4 gram	0.75	0.10	0.71	65.7%
static 3 gram	0.60	0.70	0.83	59.5%
static 3 gram	1.10	1.25	0.57	72.3%
static 4 gram	0.30	1.10	0.74	64.3%
static 4 gram	0.85	1.15	0.69	65.7%

Table 5.6: Parametric settings for the random scenario

Cluster ensemble based on a single threshold: The first cluster ensemble approach we evaluate clusters two samples together if their connectivity in $M_{i,j}^{opt}$ is larger than some threshold c_t, (i.e., at least c_t percentage of constituent clusterings agree that the two samples are clustered together). We vary c_t between 0 and 1 and plot the results in Figure 5.5. From the figure, it is apparent that, although this cluster ensemble achieves a high coverage of 79% (5% higher than the best coverage of any single clustering algorithm), the resulting precision is below our expectation. For instance, using a threshold of 0.5 (i.e., majority consensus), this ensemble approach achieves a precision of only 0.64 and 0.43 for the best case and random scenarios. Investigation of the resulting clusters reveals that the low precision is due mainly to the over-merging of unrelated clusters. In a single-threshold cluster ensemble, two clusters are merged if the connectivity in $M_{i,j}^{opt}$ between any pair of member samples x_i and x_j is greater than the threshold. As a result, unrelated clusters can be bridged together by a chain of samples, causing them to be incorrectly clustered and swelling our largest cluster to over 3,000 samples. To address this problem, we take a simple approach and avoid merging two clusters if the size of either cluster is larger than some limit. In our experiment, this limit is set to 200 (around half the size of the largest family). By imposing this threshold on the cluster size, the ensemble approach achieves precisions of 0.82 (best-case scenario) and 0.75 (random scenario), with 78% (best-case scenario) and 82% (random scenario) coverage. These results demonstrate that the ensemble approach can increase coverage by 10% over that of any single clustering algorithm, while simultaneously achieving accuracy close to the best among single clustering algorithms.

Cluster ensemble based on the ball algorithm: The ball algorithm improves over the single-threshold approach by attempting to find a set of data points that are close to one another while being and far from others. To search for such a set, the algorithm starts by sorting all the data points according to the total connectivity of

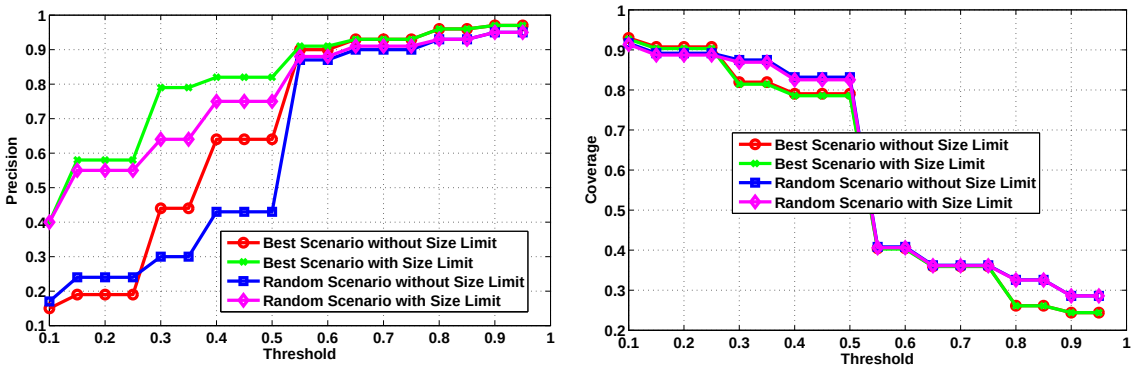


Figure 5.5: Precision and coverage of single threshold based cluster ensemble

the edges incident on the data. According to [38], this heuristic works well in practice. At each step, the algorithm finds the first unclustered data point and the set of data points that are nearby (i.e., their connectivity is larger than the threshold). These data points are considered to form a cluster, otherwise, the node forms a singleton cluster and is rejected by the clustering algorithm. In our experiment, we vary the threshold value from 0.2 to 0.9 and evaluate the effectiveness of the ball algorithm using precision and coverage. Figure 5.6 plots the experimental results. From the figure, we see that cluster ensemble using the ball algorithm performs better than the single-threshold approach, with precision consistently higher than 0.8 and coverage close to 80%. Using a threshold of 0.5, the precision for the best-case and random scenarios are 0.85 and 0.8, respectively — both very close to the maximum value among individual clusterings. Furthermore, the coverage for these two cases are 0.78 and 0.82, which are 5% and 10% higher than the best coverage for individual clusterings. Also, notice how close the cluster ensemble results are between the random and best-case scenarios, indicating that the cluster ensemble's effectiveness is not very sensitive to the choice of its constituent clusterings. This is a salient property, as it is not always possible to select the best individual clusterings.

Cluster ensemble based on the agglomerative algorithm The agglomerative algorithm is a bottom-up approach and recursively merges two nearest clusters until

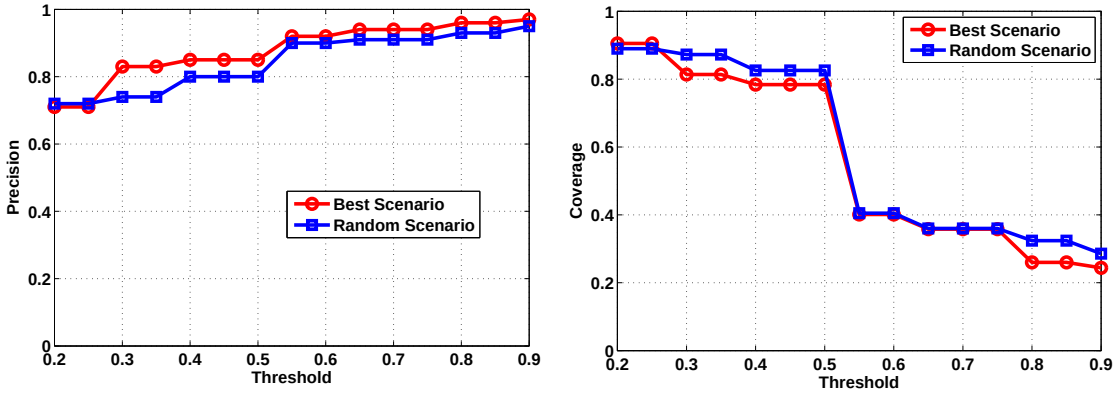


Figure 5.6: Precision and coverage of ball algorithm based cluster ensemble

the distance (defined as one minus connectivity in matrix M^{opt}) between every pair of existing clusters are larger than a certain threshold. The methods for determining the distance between clusters are called the *linkage criteria* and the commonly used linkage methods include:

- **Complete Linkage:** $D(C_1, C_2) = max\{d(c_1, c_2) : c_1 \in C_1, c_2 \in C_2\}$.
- **Single Linkage:** $D(C_1, C_2) = min\{d(c_1, c_2) : c_1 \in C_1, c_2 \in C_2\}$.
- **Average Linkage:** $D(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{c_1 \in C_1} \sum_{c_2 \in C_2} d(c_1, c_2)$.

The benefit of the agglomerative algorithm is that it starts with the most similar samples first and always continues with the "best" pair of clusters. It also allows fine-grained control in halting the merging process, such that all the remaining clusters can be far enough from each other to ensure a clear separation. In the experiment, we vary the threshold and collect results using all three different linkage methods as depicted in Figure 5.7. From the figure we can observe that single linkage is the worst of all linkage methods in terms of precision, suffering from the same over-merging problem as the single-threshold approach (i.e., two distant clusters can be bridged by a chain of samples). Average linkage is slightly better than complete linkage, resulting in 0.84 precision and 81% coverage for the random scenario and 0.87 precision and 77.6% coverage for the best-case scenario.

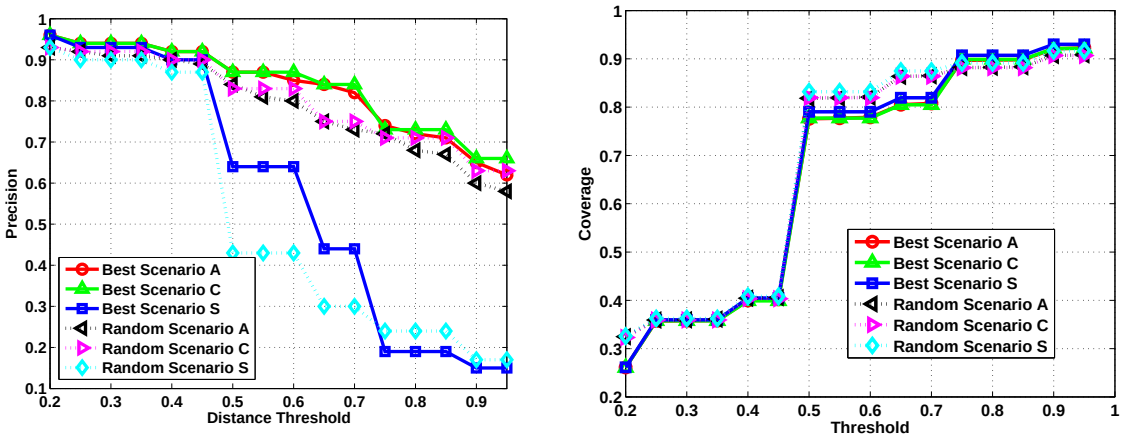


Figure 5.7: Precision and coverage of agglomerative algorithm based cluster ensemble. A, C, S in the figures represent Average, Complete and Single linkage

Cluster ensemble based on hypergraph partition Finally, we employ a more sophisticated approach and formulate the cluster ensemble problem as a hypergraph partition problem, treating the connectivity matrix as a hypergraph and attempting to find a minimum cut that divides the graph into k unconnected components while minimizing the total weights of the cut edges. In other words, the algorithm attempts to break the weakest connections between data samples, leaving only strongly connected data points to form individual clusters. There are many algorithms proposed for hypergraph partitioning. In our experiment, we make use of a hypergraph partitioning package called HMETIS [63], which exploits a multi-level partitioning algorithm and was shown to produce high-quality partitions with good scalability. We tested the HMETIS algorithm with various settings and discovered that cluster ensemble based on hypergraph partitioning has excellent coverage but suffers from low precision. The cluster ensemble using the best-case scenario can cover as many as 91.2% of all malware samples, but it achieves only 0.72 precision; results for the random scenario are 89.9% coverage with 0.71 precision. This low precision is due to a standard constraint in the graph partitioning algorithm — attempting to avoid trivial partitions by making them comparably sized. However, the malware dataset we use contains unbalanced data clusters, with the largest containing 500 samples and the

smallest comprising only 20-30. As the hypergraph partitioning approach balances the size of resulting components, it creates groups containing samples from *multiple* small families, resulting in the lower precision. Another drawback of the hypergraph-based approach is the prerequisite of specifying the number of clusters (families), which is typically hard to know *a priori*. Hence, in practice, the hypergraph-based approach may not be a good choice for cluster ensemble.

Summary In this section, we evaluated four different cluster ensemble approaches based on single threshold, balls, agglomerative and hypergraph-partitioning. Overall, we found that individual clusterings often have to make a tradeoff between precision and coverage, achieving high precision at the cost of coverage. As a result, even though some individual clusterings may excel in one aspect (precision or coverage), it often suffers in the other regard. By contrast, the cluster ensemble is able to leverage information from multiple clusterings to improve both precision and coverage. Table 5.7 summarizes the improvement of cluster ensemble over individual clusterings. The table shows that, except for hypergraph-based approach, most ensemble algorithms are able to improve precision by 5-10% and coverage by 20-40%, demonstrating its effectiveness for practical use.

5.6.4 Improving Cluster Ensemble with Cluster-Quality Measure

In this section we will evaluate the performance of cluster ensemble methods using cluster-quality measures. This experiment uses the quality measures described in Section 5.5 — i.e., Cluster Cohesion (CO) and Cluster Separation (SE). We first examine the effectiveness of these metrics in representing a cluster's quality. Next we incorporate these cluster-quality measures into each cluster ensemble approach, in an effort to improve the final clustering results.

	Best Scenario			
Ensemble Approach	Precision	Average Improvement	Coverage	Average Improvement
Single Threshold	0.82	5.38%	78.55%	26.17%
Ball Algorithm	0.85	9.24%	78.38%	25.88%
Agglomerative Algorithm	0.87	11.81%	77.60%	24.63%
Hypergraph Partitioning	0.72	-7.47%	91.20%	46.48%
Avg. Individual Clustering	0.78	N/A	62.26%	N/A
	Random Scnario			
Ensemble Approach	Precision	Average Improvement	Coverage	Average Improvement
Single Threshold	0.75	3.09%	82.54%	28.91%
Ball Algorithm	0.8	9.97%	82.56%	28.94%
Agglomerative Algorithm	0.84	15.46%	81.85%	27.83%
Hypergraph Partitioning	0.71	-2.41%	89.90%	40.40%
Avg. Individual Clustering	0.7275	N/A	64.03%	N/A

Table 5.7: Summary of cluster ensemble results and improvements over individual clusterings

5.6.5 Cluster-Quality Measures

We first examine if cluster cohesion and separation are effective measures of cluster quality. To answer this question, we take four best-coverage cases, one from each clustering (see Table 5.6.3), and compute cohesion and separation for each constituent cluster. We separate clean clusters (i.e., those consisting of malware from only one family) and mixed clusters (i..e., those consisting of malware samples from multiple families), plotting their cumulative distribution functions (CDFs) in Figure 5.8 and Figure 5.9. From the plots, we find that the two metrics work fairly well and can distinguish between clean and mixed clusters with clean clusters often having smaller cohesion and higher separation than mixed clusters. For instance, in the separation CDF, the clean-cluster line is below the mixed-cluster line, implying that a larger percentage of clean clusters have greater separation than the mixed cluster. In particular, at least 80% of the clean clusters have separation values larger than 0.97, while this is only true for 50% of mixed clusters. We find that the same argument

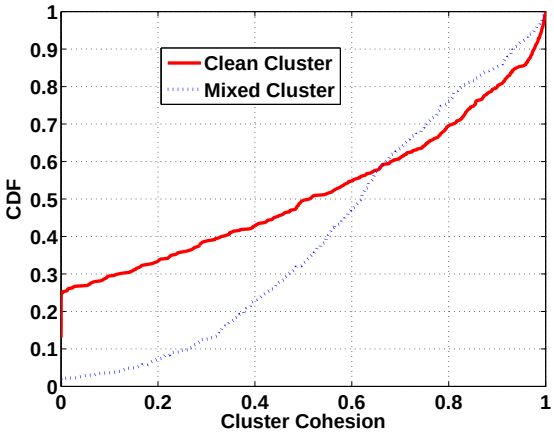


Figure 5.8: CDF for cluster cohesion

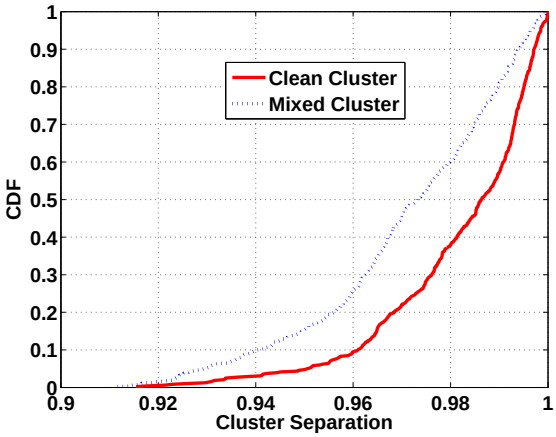


Figure 5.9: CDF for cluster separation

also holds for cluster cohesion.

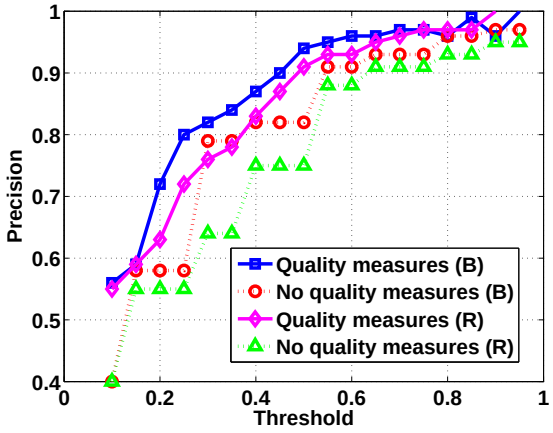
5.6.6 Cluster ensemble results with quality measures

Next, we perform experiments which integrate the cluster-quality measures into the ensemble algorithms, by augmenting the connectivity matrices according to the cluster quality (Section 5.5). We apply the same ensemble algorithms (i.e., single-threshold, ball and agglomerative) on the connectivity matrices synthesized from the same best-case (Table 5.6.3) and random (Table 5.6) scenarios, comparing the results to those generated without quality measures. Figure 5.10 (a)-(f) compare the precision and coverage results of each ensemble algorithm in both scenarios. From these figures, we observe that ensemble algorithms incorporating quality measures outperform their original counterpart, often by 5–10% in terms of precision. For example, in the agglomerative-based cluster ensemble method (Figures 5.10 (c) and (f)) with a threshold 0.5, quality measures help improve the precision from 0.87 to 0.94 for the best-case scenario and from 0.84 to 0.91 for the random scenario. A similar trend can also be observed for two other ensemble algorithms; as shown in Figures 5.10 (a)-(c), the ensemble results that incorporate quality measures are always placed at the top two lines. These first 3 plots clearly demonstrate that the quality measures

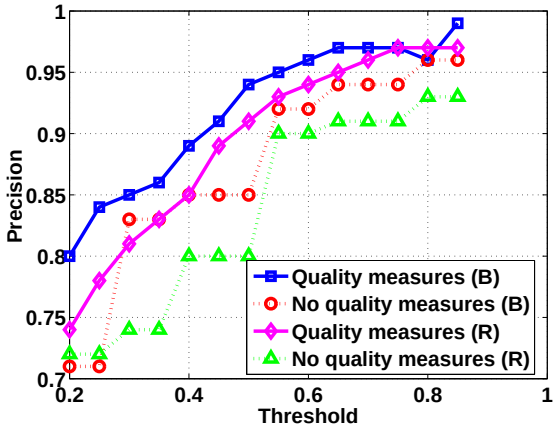
are useful in producing more accurate clusterings. On the other hand, figures 5.10 (d)-(f) show that the increase in the precision does not come without cost. We observe a decrease in the malware coverage after incorporating quality measures. This is because employing the quality measures weakens connectivity between samples in low-quality clusters, making them more likely to be excluded from final clusters. For example, in all the plots, we the coverage is shown to be reduced by 3–30%, with the biggest drop often occurring when the threshold is around 0.5. A further investigation reveals that this is due to our selection of member clusterings, (i.e., a half from dynamic approaches and a half from static approaches). Often, when malware samples are mis-clustered by a behavioral or static clustering approach, other approaches of the same type similarly mis-cluster the samples. In our experiments, we use an equal amount of behavioral and static clusterings as input into ensemble methods. This combined with a threshold of 0.5 (which corresponds to a majority vote among clusterings) can result in greater coverage at the expense of precision due to the wrong consensus of approaches from the same type. Fortunately, by incorporating quality measures, the reduced connectivity between samples with disagreeing approaches drop below the threshold, lowering the sample coverage as show in Figure 5.10. However, the decrease in the coverage is not necessarily bad, as the remaining clusters are often of better quality, implying higher confidence in the resulting malware groups. Incorporating more diversified sets of analysis techniques and clustering algorithms may help mitigate the problem, but we leave this as our future work.

5.7 Related Work

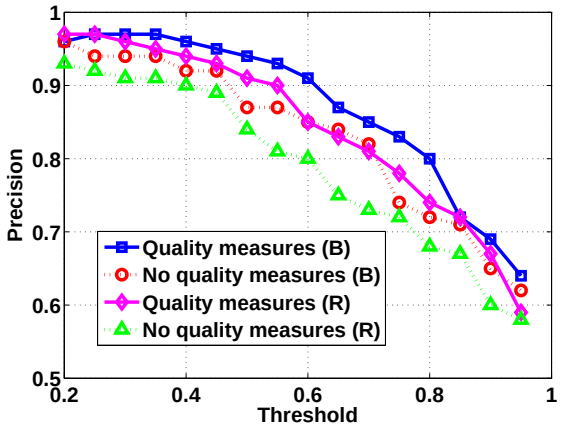
As an overwhelming number of new malware programs created everyday have already outpaced the existing analysis techniques and made the manual analysis infeasible, automatic malware clustering and classification has become a very active research subject in both the security community and industry. Various approaches



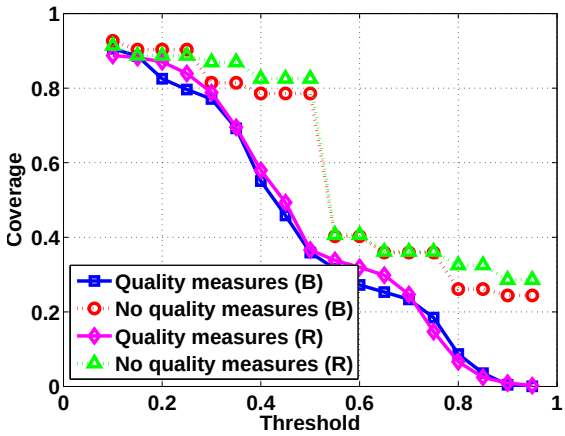
(a) Ensemble results based on the single-threshold algorithm



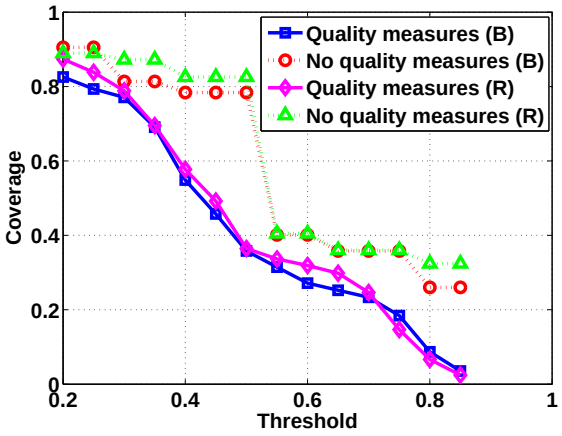
(b) Ensemble results based on the ball algorithm



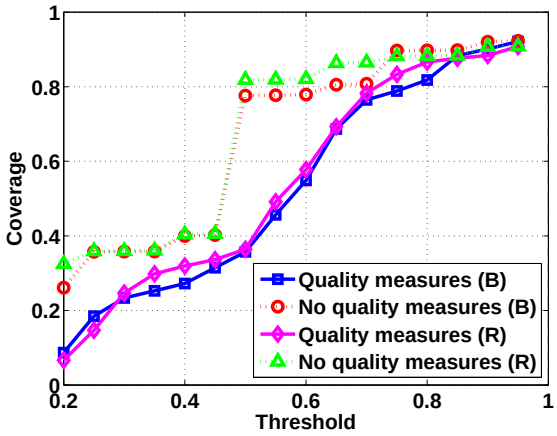
(c) Ensemble results based on the agglomerative algorithm



(d) Ensemble results based on the single-threshold algorithm



(e) Ensemble results based on the ball algorithm



(f) Ensemble results based on the agglomerative algorithm

Figure 5.10: Cluster ensemble results with cluster-quality measures. In the figure (B) represents the best case scenario and (R) represents the random case scenario

have been developed and applied in malware clustering. They can naturally be divided into two categories based on the type of malware features they rely on. Static feature based approaches are among the first few approaches proposed for analysis and detection of malware programs, due mainly to their ease of implementation and fast speed of analysis. In static feature based approaches, malware binaries are analyzed either manually or automatically through tools, such as disassembler or decompiler. Features like PE headers, raw binary size, import tables [81, 113], code patterns [22], instruction sequences [53] or binary sequences [50] are extracted and used to train a wide range of learning algorithms such as naive Bayes, decision trees, SVM, etc. [56] for malware categorization and detection. Unfortunately, the popularity of static approaches encouraged malware authors to develop various obfuscation techniques that thwart the static analysis [68, 84]. Most popular obfuscation methods include packing, polymorphism, and metamorphism. However, because these low-level obfuscation only changes the syntax of malware programs but keeps their semantics intact, defenders came up with dynamic analysis technologies that detect malicious codes based on semantic features. Recently, dynamic malware analysis approaches have received significant attention, and several commercial and open systems have also been implemented such as CWSandbox [70], Anubis [3], Norman Sandbox [78]. These systems execute malware programs in the instrumented or virtual environments to monitor their run-time behavior. Taking advantage of these systems for dynamic feature extraction, researchers have successfully applied a wide range of clustering/classification algorithms for malware categorization. For instance, Gheorghescu [37] measured the distance of basic block sequences between existing and unknown samples to simplify the detection of new malware. Lee and Mody [64] employed a nearest-neighbor approach to find malware programs with similar behaviors in terms of system events, such as registry and file system modification. Rieck *et al.* [86] trained SVM classifier using the malware's run-time behavior, such as copy files and create processes, to

learn and classify unknown samples. Bailey *et al.* [10] employed a hierarchical clustering algorithm to group similarly-behaving malware samples based on non-transient state changes malware caused to the system. More recently, Bayer *et al.* [13] adopted locality-sensitive hashing (LSH) and Rieck *et al.* [88] developed a prototype-based clustering algorithm to reduce the runtime complexity and enable the behavior-based clustering algorithm to scale to a large number of malware samples. Unfortunately, the dynamic analysis is not without its own limitations. Like static approaches, as the attackers become aware of the prominence of dynamic analysis, they start to develop countermeasures. For instance, because many of the dynamic analysis approaches rely on virtual machines to provide a safe environment for running malware programs, an increasing number of malicious codes are carrying code snippets that detect the existence of virtual machine monitors [69, 80]. Usually, when malicious codes detect the VM, they either terminate immediately or shut off malicious functionality [12] such that the full spectrum of their malicious behavior cannot be observed. Another inherent limitation of dynamic analysis is that it can only observe a single code path that is executed during a particular run, which may lead to an incomplete picture of malware activities. In particular, for botnets and many other trigger-based malware, without being provided with the exact commands or environmental settings, they hardly exhibit the real malicious behaviors embedded in their codes. Fortunately, static approaches do not suffer from this limitation and thus provide a unique benefit of covering all the code paths. Therefore, a systematic approach to integrating static and dynamic analysis algorithms and combining their respective strengths will be highly desirable. To achieve this goal, this chapter presented `DUET`, a system that utilizes cluster ensemble techniques to efficiently integrate dynamic and static feature based clusterings.

Cluster ensemble is a process of obtaining a single consensus and better-performing clustering result from a number of different clusterings [36, 95]. Because different

clustering algorithms have different perspectives and assumptions of a given data set, the cluster ensemble can exploit the strengths from individual clusterings to provide improved overall partition of the given data in terms of robustness, stability and confidence. Several approaches have been proposed and successfully used for clustering combinations. For instance, Strehl and Ghosh [96] considered cluster ensemble as a knowledge reuse framework for combining different clusterings and proposed three different consensus functions, i.e., Cluster-based Similarity Partitioning Algorithm (CSPA), Hypergraph Partitioning Algorithm (HGPA) and Meta Clustering Algorithm (MCLA). Fern and Brodley [31] also modeled the cluster ensemble as a graph partitioning problem and addressed it using Hybrid Bipartite Graph Formulation (HBGF) algorithm. Hong et al. [46] implemented an approach that combines relabeling and voting to achieve the best agreement between the labels of partitions. Topchy *et al.* [102] proposed to construct a consensus function based on information-theoretic principles using generalized mutual information (MI) between the empirical probability distribution of labels in the consensus partitions. Azimi *et al.* [9] developed a new ensemble method which creates a new feature space from initial clustering outputs and used k-means in the new feature space to generate final results. Another commonly used method in cluster ensemble is to base the consensus function on the co-association matrix which represents the association or connectivity between each pair of data samples. For example, Fed [34] and Jain [33] applied fixed threshold and single linkage hierarchical clustering to the co-association matrix. The validity of their algorithm is compared with the standard k-means clustering. In summary, clustering ensembles have emerged as a prominent way to improve robustness, stability and accuracy of clustering. In this chapter, we employ simple yet effective cluster ensemble algorithms, i.e., co-association (connectivity) matrix based approaches, and demonstrate that they can achieve both higher precision and better coverage of malware samples than individual clustering algorithms alone. A more complex cluster

ensemble algorithm such as mutual information theory can easily be incorporated into DUET and test its effectiveness.

5.8 Concluding Remarks

In this chapter, we design, develop and evaluate an automatic malware-clustering system, called DUET, for grouping malware samples into families sharing common traits in terms of both static-code and dynamic behavior-features. We first build a dynamic malware-analysis system that automatically executes malware programs inside a sandboxed virtual environment and collects their runtime system call traces. We apply a prototype-based clustering algorithm on the collected traces, showing that the dynamic approach can successfully cluster around 70% of all malware samples $\approx$ 75% precision. Comparing these results with static-feature based clustering in MutantX , we confirm that, due to their respective limitations, static and dynamic approaches cover different sets of malware sample providing a strong motivation to develop a new system, DUET, that effectively combines their strengths. DUET, using cluster ensemble methods, integrates multiple clustering results from both dynamic- and static-based approaches. Evaluating with real-life malware samples, we have shown that cluster ensemble methods are quite effective in improving the overall precision and coverage of malware clustering. We assess the performance of ensemble methods on both best-case scenarios and random scenarios, demonstrating that it can improve the coverage by 20–50% while achieving nearly the maximum precision of the individual clustering algorithms. Finally, we further improve the existing cluster ensemble algorithms by taking advantage of cluster-quality measures. The intuition is that high-quality clusters (i.e., whose samples are strongly connected to each other) should more likely be preserved in the final clustering results. Thus, we exploit cluster cohesion and separation as metrics for measuring cluster quality, adjusting the weights between samples in the connectivity matrix according to the quality of

their clusters. Experiment results show that using cluster quality can further improve precision by 5–10%, approaching 0.9. However, this improvement comes at the cost of reducing the cluster coverage, mainly because of the higher standard imposed on the quality of final clusters. These results demonstrate that cluster ensemble methods are effective at integrating multiple malware analysis techniques, allowing `DUET` to achieve automatic and efficient malware analysis and help ensure timely defense against emerging malware threats.

CHAPTER VI

Conclusions

As malware threats have evolved from a hobby of hackers into powerful arsenals for cyber-criminals to illegally gain profit, the amount of malware programs has increased beyond the capabilities of existing analysis techniques. Typically, an AV company receives more than ten thousand new suspicious samples daily, which is simply too many for manual analysis. This delays responses to new threats, granting malware writers a sufficient time window to roll out new malware variations and harm users. In the light of this observation, the primary goal of this dissertation is to improve the automation and scalability of malware analysis techniques, with a particular emphasis on automating portions of the malware-processing workflow traditionally performed by human experts. This dissertation research makes the following contributions.

In Chapter II, we investigated a novel malware database management system called `SMIT` that addresses the challenge of automatically determining if incoming suspicious samples are indeed malicious. `SMIT` exploits the insight that most new samples are simple syntactic variations of existing malware. Thus, one way to ascertain the maliciousness of a sample is to check if it is sufficiently similar to any known malware program. `SMIT` can efficiently make such decisions based on the malware's function-call graph, a high-level structural representation that is less susceptible to the low-level obfuscations employed by malware writers to evade detection. Because

each malware program is represented as a call graph, searching for similar malware programs becomes a nearest-neighbor search problem in a graph database. To address the scalability issue associated with graph comparison, the dissertation proposed the Neighbor-Biased Hungarian Algorithm (NBHA), which optimizes a polynomial-time graph-similarity algorithm by exploiting common sub-structures in malware call graphs. The results of NBHA closely approximate the inter-graph edit distance while reducing the computational complexity to $O(n^3)$. Furthermore, the dissertation developed a multi-resolution indexing scheme to solve the scalability issue related to the graph database search. The indexing scheme uses a computationally economical feature vector for early pruning and then resorts to a more accurate, but computationally more expensive, graph similarity function when it needs to pinpoint the most similar neighbors. The unique combination of these techniques affords `SMIT` significant pruning power and allows it to easily scale to support hundreds of thousands of malware samples, with the potential to handle millions.

In Chapter III, we proposed a novel malware-clustering framework, called `MutantX` , designed to help malware analysts automatically derive labels for unknown malware samples. `MutantX` clusters malware samples according to the similarity of their machine code instructions and automatically generates labels using their cluster association. The underlying intuition is that malware programs sharing significant similarity are potentially derived from the same code base and are thus likely to come from the same malware family. By grouping similar samples into clusters, the entire cluster can be labeled accurately by analyzing only a few representative samples from each cluster. To address the challenge of efficient feature extraction, we developed an encoding scheme that exploits the IA-32 instruction format, encoding each malware program into a sequence of uniform opcodes (operation codes). By ignoring the operands in machine instructions, the encoding scheme effectively captures the semantics of the instruction while being resilient to low-level mutation resulting from

obfuscation or address relocation. Next, an n-gram analysis is applied to the opcode sequence, constructing a feature vector. To further improve the scalability of the clustering algorithm and reduce the memory footprint, we exploit a hash trick that reduces the high-dimensional feature vector into a lower-dimensional feature space. The similarity between two malware samples can then be computed as the Euclidean distance between their corresponding feature vectors, which serves as an input into a prototype-based clustering algorithm determining the proper grouping of malware.

After identifying the malware samples' family, the next step is to generate AV signatures that can be deployed to end users, protecting them from newly discovered malware. In Chapter IV, motivated by the signature-explosion problem current;u facing AV industries, we studied efficient ways to automatically generate string signatures. A string signature is a contiguous byte sequence meant to match variants of a malware family, rather than a specific malware program, and can thus reduce the size of the signature database. Unfortunately, most string signatures used today are created manually, which is slow and labor-intensive. To address this problem, we developed `Hancock`, the first-known automatic system for string-signature generation. At its foundation, the `Hancock` system is a set of algorithms that efficiently filter out potential false-positive signatures. First, we built a Markov chain model, trained on a large benign-program set, that accurately estimates the occurrence probability of a candidate signature in benign programs. This model allows `Hancock` to eliminate the majority of false-positive signatures without expensive scanning through a benign-program database. In addition, we observe that many false-positive signatures are resulted from standard library functions or generic code sequences shared across both malicious and benign programs. Consequently we devised a set of content-aware selection heuristics and diversity-based filtering techniques that check if a candidate belongs to a library function or some common code base. Together, these techniques provide `Hancock` the strong discrimination power necessary to automatically create

high-quality string signatures. We evaluated `Hancock` on a large real-world malware programs, showing that it can generate signatures with false-positive rates below 0.1%.

Finally, this dissertation studied the strengths and weaknesses associated with the two major malware analysis approaches (i.e., static and behavioral), proposing an effective system to combine them for better results. In Chapter V, we first built a dynamic malware-monitoring system that executes malware programs in a controlled virtual environment, collects their runtime API traces, and converts those traces into feature vectors for processing by existing clustering algorithms. The results demonstrated that static and dynamic approaches tend to work well for different types of malware programs, indicating its potential for improving overall clustering performance by leveraging the strengths of different approaches. To achieve this goal, we employed cluster ensemble methods, a process of obtaining a single consensus and better-performing clustering results from a number of different clusterings. Applying cluster ensemble to several static and dynamic clustering results, we demonstrated its ability to improves clustering accuracy and successfully group malware programs that cannot be analyzed by any single approach,increasing coverage by 20–40%. Finally, we make a further improvement over existing ensemble methods, by incorporating cluster-quality measures into ensemble algorithms. By preserving strongly connected samples in high-quality clusters, this new approach increases the cluster precision by 5–10% with enhanced quality of final clusters.

There are a number of areas one can pursue to extend the large-scale malware analysis in this dissertation as follows.

- **Automatic generation of malware behavioral signatures** Signature-based detection is lagging behind the generation of malicious threats, suggesting that security technologies relying on signatures should be complemented with additional heuristics, such as behavior-based detection. Behavior-based detection

identifies actions performed by malware rather than syntactic signatures. As a result, it is more resilient to malware polymorphism and obfuscation, and it has the potential to capture an entire malware family with a single behavioral signature. Unfortunately, identifying distinguishing malicious actions and transforming them into usable behavioral signatures are a complex process. Currently, this is primarily done by security experts, requiring significant experience and time. As a result, an efficient system for analyzing malware run-time behavior and automatically extracting behavioral signatures will be highly beneficial in combating malware threats. Building such a system entails several key challenges in the monitoring, formal modeling and extraction of malicious behaviors as well as in developing efficient filters to minimize false-positive signatures. The work on string signature generation in this thesis can potentially be extended to address these problems.

- **Mobile malware detection and containment** Modern mobile devices, equipped with increased resources and high connectivity, are becoming more intelligent and functionally complex. However, with these advanced capabilities, mobile devices are increasingly exposed to malware and malicious attacks, such as sensitive information theft, location privacy risks, etc. For instance, researchers have recently demonstrated a vulnerability in Android OS that allows rootkits to be installed, potentially leading to a mobile botnet. In 2010, it is estimated that over 100,000 Android users were affected by mobile malware, forcing Google to pull 21 malicious applications from its market [58]. As a result, it will be necessary to develop efficient malware analysis systems for mobile environments, facilitating the understanding, classification and mitigation of new-generation mobile malware.

- **Monitoring and characterizing malware behaviors in online social net-**

works and clouds The massive use of online social networks has made them an increasingly attractive target for malicious activities. In particular, botnets (e.g., Koobface) have already leveraged the implied trust within social networks to increase their likelihood of successful social engineering attacks, such as spamming and malware propagation. Investigation into the unique features of malware propagation in these networks will help us gain insight into their behaviors. For example, learning techniques could be used to derive patterns from the malicious code posted on social networking sites and such patterns may serve as signatures to proactively blacklist future malware sites and protect users. In addition, properties unique to social networks, such as degree distribution, friend relationships and communication patterns, can be explored to obtain a deeper understanding of social malware and anticipate the next phase of their evolution.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Margareta Ackerman and Shai Ben-David. Measures of Clustering Quality: A Working Set of Axioms for Clustering. In *Proceedings of the 22nd Annual Conference on Neural Information Systems Processing*, 2008.

[2] Hyang ah Kim. Autograph: Toward automated, distributed worm signature detection. In *In Proceedings of the 13th Usenix Security Symposium*, pages 271–286, 2004.

[3] Anubis. Analyzing unknown binaries. http://anubis.iseclab.org/, 2010.

[4] Austin Appleby. Murmurhash 2.0. http://sites.google.com/site/murmurhash/.

[5] William Arnold and Gerald Tesauro. Automatically generated win32 heuristic virus detection. In *In Proceedings of VIRUS BULLETIN CONFERENCE*, 2000.

[6] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the twenty-second annual symposium on Computational geometry*, SCG '06, pages 144–153, New York, NY, USA, 2006. ACM.

[7] ASpack Software. Aspack. http://www.aspack.com/.

[8] ASPack Software. ASProtect. http://www.aspack.com/, 2008.

[9] Javad Azimi, Monireh Abdoos, and Morteza Analoui. A new efficient approach in clustering ensembles. In *Proceedings of the 8th international conference on Intelligent data engineering and automated learning*, IDEAL'07, pages 395–405, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Michael Bailey, Jon Andersen, Z. Morley mao, and Farnam Jahanian. Automated classification and analysis of internet malware. Technical report, In Proceedings of Recent Advances in Intrusion Detection (RAID07, 2007.

[11] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *RAID*, pages 178–197, 2007.

[12] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, CA, USA*, 02 2010.

[13] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[14] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security*, 2009.

[15] Stefano Berretti, Alberto Del Bimbo, and Enrico Vicario. Efficient matching and indexing of graph models in content-based retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(10):1089–1105, 2001.

[16] BindView. Strace. http://razor.bindview.com/tools/desc/strace_readme.html, 2005.

[17] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *In Proc. ACM SIGMOD International Conference on Management of Data*, 1997.

[18] Ismael Briones and Aitor Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.

[19] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Ero Carrera and Gergely Erdelyi. Digital genome mapping advanced binary malware analysis. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.

[21] Tzi-cker Chiueh. Content-based image indexing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 582–593, 1994.

[22] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[23] Clam AntiVirus. Creating signatures for ClamAV. www.clamav.net/doc/latest/signatures.pdf, 2007.

[24] Peter Coogan. Spyeye bot versus zeus bot. http://www.symantec.com/connect/blogs/spyeye-bot-versus-zeus-bot, 2010.

[25] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, New York, NY, USA, 2005. ACM.

[26] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 252–266, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Inc. Damballa. 3% to 5% of enterprise assets are compromised by bot-driven targeted attack malware. http://www.damballa.com/downloads/press/Failsafe_3_%28PR_FINAL_2009-3-2%29.pdf, 2008.

[28] Shagnik Das, Anand Mistry, Diana Negoescu, Georgina Reed, and Sudhir Kumar Singh. A graph matching problem. Techical report, IPAM Research in Industrial Projects for Students (RIPS), 2008.

[29] Thomas Dullien, Rolf Rolles, and Ruhr universitaet Bochum. Graph-based comparison of executable objects. In *University of Technology in Florida*, 2005.

[30] Nicolas Falliere. Stuxnet introduces the first known rootkit for industrial control systems. Technical report, Symantec Corporation, 2010.

[31] Xiaoli Z. Fern, Carla E. Brodley, Xiaoli Zhang Fern, and Carla E. Brodley. Solving cluster ensemble problems by bipartite graph partitioning. In *In Proceedings of the International Conference on Machine Learning*, 2004.

[32] Halvar Flake. Structural comparison of executable objects. In *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*, pages 161–173, 2004.

[33] A. L. N. Fred and A. K. Jain. Data clustering using evidence accumulation. In *Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 4 - Volume 4*, ICPR '02, pages 40276–, Washington, DC, USA, 2002. IEEE Computer Society.

[34] Ana Fred. Finding consistent clusters in data partitions. In *In Proc. 3d Int. Workshop on Multiple Classifier*, pages 309–318. Springer, 2001.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[36] R. Ghaemi, N. Sulaiman, H. Ibrahim, and N. Mustapha. A Survey: Clustering Ensembles Techniques. *PROCEEDINGS OF WORLD ACADEMY OF SCIENCE, ENGINEERING AND TECHNOLOGY*, 38, February 2009.

[37] Marius Gheorghescu. An automated virus classification system. In *Proceedings of VIRUS BULLETIN CONFERENCE OCTOBER 2005*, 2005.

[38] Aristides Gionis, Heikki Mannila, and Panayiotis Tsaparas. Clustering aggregation. *ACM Trans. Knowl. Discov. Data*, 1, March 2007.

[39] T. Gonzalez. Clustering to minimize the maximum intercluster distance. In *Theoretical Computer Science*, volume 38, pages 293–306, 1985.

[40] Herv Debar Grgoire Jacob1 and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3), 2008.

[41] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *RAID '08*, pages 98–115, 2008.

[42] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, 2005.

[43] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, 2009.

[44] Huahai He and Ambuj K. Singh. Closure-tree: An index structure for graph queries. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 38, 2006.

[45] Hex-rays. The IDA Pro Disassembler and Debugger. http://www.hex-rays.com/idapro/, 2008.

[46] Yi Hong, Sam Kwong, Yuchou Chang, and Qingsheng Ren. Unsupervised feature selection using clustering ensembles and population based incremental learning algorithm. *Pattern Recognition*, 41(9):2742–2756, 2008.

[47] Xiaohua Hu and Illhoi Yoo. Cluster ensemble and its applications in gene expression analysis. In *Proceedings of the second conference on Asia-Pacific bioinformatics - Volume 29*, APBC '04, pages 297–302, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[48] Ilfak Guilfanov. Fast Library Identification and Recognition Technology. http://www.hex-rays.com/idapro/flirt.htm, 1997.

[49] Computer Security Institute. 12th annual edition of the csi computer crime and security survey. Technical report, Computer Security Institute, 2007.

[50] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Fast, scalable code reuse detection in binary code. tech report CMU-CyLab-10-006, Carnegie Mellon University, 2009.

[51] Jibz, Qwerton, snaker, and xineohP. Peid 0.95. http://www.peid.info/, 2008.

[52] Derek Justice. A binary linear programming formulation of the graph edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(8):1200–1214, 2006. Fellow-Hero,, Alfred.

[53] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *JOURNAL IN COMPUTER VIROLOGY*, 1:13–23, 2005.

[54] Jeffrey O. Kephart and William C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, 1994.

[55] KilianWeinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26 th International Conference on Machine Learning*, 2009.

[56] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2006, 2006.

[57] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, 2006.

[58] Tom Krazit. Mobile malware continues to rise with android users as targets. http://moconews.net/article/419-mobile-malware-continues-to-rise-with-android-users-as-targets/, 2011.

[59] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, January 2004.

[60] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *In RAID*, pages 207–226. Springer-Verlag, 2005.

[61] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[62] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 1955.

[63] Karypis Lab. Family of graph and hypergraph partitioning software. http://glaros.dtc.umn.edu/gkhome/views/metis, 2010.

[64] Tony Lee and Jigar J.Mody. An automated virus classification system. In *Proceedings of VIRUS BULLETIN CONFERENCE OCTOBER 2005*, 2005.

[65] Zhichun Li, Manan Sanghi, Yan Chen, Ming yang Kao, and Brian Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *In SP 06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (Oakland06*, pages 32–47. IEEE Computer Society, 2006.

[66] Zhenkai Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 215–224, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, New York, NY, USA, 2005. ACM.

[68] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly, 2003.

[69] Tom Liston. On the cutting edge: Thwarting virtual machine detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, 2006.

[70] malwareanalysis.org. Cwsandbox:: Behavior-based malware analysis. http://mwanalysis.org/, 2011.

[71] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[72] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *In Proceedings of the Annual Computer Security Applications Conference (ACSAC*, 2007.

[73] M.Christodorescu, S.Jha, S.A.Seshia, D.Song, and R.E.Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[74] Richard Myers, Richard C. Wilson, and Edwin R. Hancock. Bayesian graph edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(6), 2000.

[75] Michel Neuhaus and Horst Bunke. An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In *SSPR/SPR*, pages 180–189, 2004.

[76] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.

[77] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[78] Norman. Norman sandbox information center. http://www.norman.com/security_center/security_tools/, 2010.

[79] Gunter Ollmann. Serial variant evasion tactics techniques used to automatically bypass antivirus technologies. http://www.damballa.com/downloads/r_pubs/WP_SerialVariantEvasionTactics.pdf, 2010.

[80] Alfredo Andres Omella. Methods for virtual machine detection. http://www.s21sec.com/descargas/vmware-eng.pdf, 2009.

[81] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recogn. Lett.*, 29:1941–1946, October 2008.

[82] Matt Pietrek. An In-Depth Look into the Win32 PE File Format. http://msdn.microsoft.com/en-us/magazine/cc301805.aspx, 2002.

[83] Marco Pontello. Trid v2.02. http://mark0.net/soft-trid-e.html, 2008.

[84] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007. USENIX Association.

[85] Jason Raber and Eric Laspe. Deobfuscator: An automated approach to the identification and removal of code obfuscation. *Reverse Engineering, Working Conference on*, 0:275–276, 2007.

[86] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[87] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dssel, and Pavel Laskov. Learning and classification of malware behavior. In Diego Zamboni, editor, *DIMVA*, volume 5137 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2008.

[88] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. tech report, Berlin Institute of Technology, 2009.

[89] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Graph-Based Representations in Pattern Recognition*, volume 4538, 2007.

[90] Ran El-Yaniv Ron Begleiter and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:384–421, 2004.

[91] Dennis Shasha, Jason, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.

[92] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[93] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, Alex Strehl, and Vishy Vishwanathan. Hash kernels. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2009.

[94] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[95] Alexander Strehl, Joydeep Ghosh, and Claire Cardie. Cluster ensembles - a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, 3:583–617, 2002.

[96] Alexander Strehl, Joydeep Ghosh, and Claire Cardie. Cluster ensembles - a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, 3:583–617, 2002.

[97] Symantec Corp. Symantec Global Internet Security Threat Report. Volume XII. http://www.symantec.com/, April 2008.

[98] Yong Tang and Shigang Chen. Defending against internet worms: A signature-based approach. In *In Proceedings of IEEE INFOCOM05*, 2005.

[99] The Silicon Realms Toolworks. Armadillo. http://www.siliconrealms.com/armadillo.htm, 2008.

[100] Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[101] T.Lee and J.J.Mody. Behavioral classification. http://www.microsoft.com/downloads/details.aspx?FamilyID =7b5d8cc8-b336-4091-abb5-2cc500a6c41a&displaylang=en,2006.

[102] Alexander Topchy, Anil K. Jain, and William Punch. Combining multiple weak clusterings. *Data Mining, IEEE International Conference on*, 0:331, 2003.

[103] Trend Threat Research Team. Zeus: A persistent criminal enterprise. http://us.trendmicro.com/imperia/md/content/us/trendwatch/researchandanalysis/zeusaper 2010.

[104] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. *Reverse Engineering, Working Conference on*, 0:45–54, 2005.

[105] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[106] UPX. the ultimate packer for executables. http://upx.sourceforge.net/.

[107] VirusTotal. Free online virus, malware and url scanners. http://www.virustotal.com/, 2010.

[108] VMProtect. Vmprotect. http://www.vmprotect.ru/, 2008.

[109] VMWare. Vix api. http://www.vmware.com/support/developer/vix-api/, 2010.

[110] VxHeaven. Vxheaven virus collection. http://vx.netlux.org/, 2010.

[111] Helen J. Wang, Helen J. Wang, Chuanxiong Guo, Chuanxiong Guo, Daniel R. Simon, Daniel R. Simon, Alf Zugenmaier, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *In ACM SIGCOMM*, pages 193–204, 2004.

[112] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: black-box exploit detection and signature generation. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 37–46, New York, NY, USA, 2006. ACM.

[113] Georg Wicherski. pehash: A novel approach to fast malware clustering. In *2nd Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET'09)*, 2009.

[114] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 223–234, New York, NY, USA, 2005. ACM.

[115] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 335–346, 2004.

[116] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005.

[117] Mark Vincent Yason. The art of unpacking. https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf, 2007.

[118] Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 95–104, New York, NY, USA, 2010. ACM.

[119] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2005. USENIX Association.

[120] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search, 1999.

[121] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1993.

[122] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*. Springer, 2006.

[123] Pavel Zezula, Paolo Ciaccia, and Fausto Rabitti. M-tree: A dynamic index for similarity queries in multimedia databases. Technical Report 7, HERMES ESPRIT LTR Project, 1996.

[124] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[125] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: tree + delta $\leq$ graph. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 938–949, 2007.