UNIVERSITY of CALIFORNIA
Santa Barbara

A Signal Processing Approach To Malware Analysis

A dissertation submitted in partial satisfaction of the
requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Lakshmanan Nataraj

Committee in charge:

Professor B. S. Manjunath, Chair
Professor Kenneth Rose
Professor Shivkumar Chandrasekaran
Professor Giovanni Vigna
Professor Christopher Kruegel

December 2015

The dissertation of Lakshmanan Nataraj is approved.

Professor Kenneth Rose

Professor Shivkumar Chandrasekaran

Professor Giovanni Vigna

Professor Christopher Kruegel

Professor B. S. Manjunath, Committee Chair

August 2015

A Signal Processing Approach To Malware Analysis

To Amma and Appa

Acknowledgements

First I would like to thank Amma and Appa. Without their dedication and countless sacrifices, I will not have reached this stage in life. I owe it to my spiritual Guru, the Jagadguru of Sringeri, whose blessings have immensely impacted me in every sphere of my life. Finally, for a believer, there are certain moments when one can realize the power of God. I am blessed to have cherished many of those moments.

I would also like to thank Lakshmi and Sharath for their constant support, Prabha for her encouragement and Aparna for her affection.

I would like to thank Prof. Manjunath for taking me into his lab right from an early stage when I was still a new Master's student. During my initial years, I was lucky to work on lots of interesting projects in Digital Forensics and Data Hiding. Perhaps, that is what motivated me to encode a hidden message in this thesis (*Sarvam Sattva Mayam*). During the start of my PhD, Prof. Manjunath encouraged me to think of how Signal Processing techniques can be applied to challenging problems in Computer Security. While I was struggling to learn the new field of Computer Security, his advice to me was to think outside the box and come up with methods that are well known in Signal Processing but less widely known in Security. This advice helped a lot in shaping my way of approaching problems, which eventually led to this thesis. Prof. Manjunath also gave me a lot of personal advice ranging from how to give presentations to public speaking to personality development and lots more. He was extremely supportive when I had a personal emergency and had to leave the lab for more than 6 months. Prof. Manjunath is well known for his patience. There were numerous occasions when I may have annoyed him or given him some problem but during all those times, he was extremely patient. There are many things I still need to learn from him such as his

patience, dedication to work, thinking outside the box, taking timely calculated risks, detachment to results and many more.

I would like to thank Prof. Giovanni Vigna for being a part of my committee and supporting me in numerous occasions. Giovanni helped me a lot during my initial years when I was new to the field of computer security. His critical suggestions during many of our project meetings played an important role in the way I think about problems and solutions. I would like to thank Prof. Christopher Kruegel for being a part of my committee. Though I only had a few meetings with Chris, his suggestions were very crucial. He knew where exactly I was going wrong and would explain it such a way that I myself would easily realize where I had gone wrong. I would like to thank Prof. Shivkumar Chandrasekaran for being a part of my committee and for many suggestions during meetings. I would also like to thank Prof. Ken Rose for being a part of my committee and for his many valuable comments about my approach.

I have been fortunate to work with many colleagues. I thank Anindya for mentoring me during my initial years. Gregoire helped me a lot when I was new to the field of Computer Security. I cherish the interactions I had with Dhilung. Karthikeyan was very helpful during crucial times of my PhD. I have had the pleasure of having many wonderful lab members who I would like to thank for being a part of my journey: Zefeng, Pratim, Amir, Jiejun, Karthikeyan, Santhosh, Vignesh, Utkarsh, Diana, Carlos, Arhcith, Nilofar, Po-Yu, Chris Wheat, Hui, Thuyen, Luca, Marco, Ken, Kaushal and many more. There are also numerous friends outside the lab who have been an important part of my graduate student life and I would like to thank them all.

I enjoyed the time during my internships at Walt Disney Animation Studios and SRI International. I thank Joe Marks and Mark Dawson of Disney for their constant

encouragement. I really enjoyed working with Vinod Yegneswaran and Phil Porras at SRI and learnt a lot from them.

Finally I thank my funding agency, Office of Naval Research (ONR) for supporting me during my Phd. I would like to specially thank Dr. Sukarno for giving me a lot of feedback when we met at conferences.

Curriculum Vitæ

Lakshmanan Nataraj

Sep. 2015 **Doctor of Philosophy**
Department of Electrical and Computer Engineering
University of California, Santa Barbara

Dec. 2009 **Master of Science**
Department of Electrical and Computer Engineering
University of California, Santa Barbara

Jun. 2007 **Bachelor of Engineering**
Department of Electronics and Communication
Sri Venkateswara College of Engineering, Sriperumbudur, Tamil Nadu, India

Field of Study

Image Processing, Malware Analysis

Experience

2008 - 2015 **Graduate Student Researcher**
Vision Research Lab, University of California, Santa Barbara, CA

2011 **Summer Research Associate**
SRI International, Menlo Park, CA

2008 **Summer Research Associate**
Walt Disney Animation Studios, Burbank, CA

Publications

Lakshmanan Nataraj, S Karthikeyan, B.S. Manjunath, SATTVA: SpArsiTy inspired classificaTion of malware VAriants, ACM Workshop on Information Hiding and Multimedia Security (IH and MMSEC), Portland, Oregon, Jun. 2015

Lakshmnanan Nataraj, Dhilung Kirat, B.S. Manjunath and Giovanni Vigna, SARVAM: Search And RetrieVAl of Malware, Annual Computer Security Applications Conference (ACSAC) Workshop on Next Generation Malware Attacks and Defense (NGMAD), New Orleans, Dec. 2013

Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna and B.S. Manjunath, SigMal: A Static Signal Processing Based Malware Triage, Annual Computer Security Applications Conference (ACSAC), Dec. 2013

Lakshmanan Nataraj, Vinod Yegneswaran, Phil Porras, Jian Zhang, A Comparative Assessment of Malware Classification using Binary Texture Analysis and Dynamic Analysis, Workshop on Artificial Intelligence and Security (AISec), Chicago, Oct. 2011

Lakshmanan Nataraj, S. Karthikeyan, Gregoire Jacob, B.S. Manjunath, Malware Images: Visualization and Automatic Classification, International Symposium on Visualization for Cyber Security (VizSec) , Jul. 2011

L. Nataraj, A. Sarkar and B. S. Manjunath, Precise Localization Of Key-points To Identify Local Regions For Robust Data Hiding, IEEE International Conference on Image Processing, Sep. 2010

Lakshmanan Nataraj, Gregoire Jacob, B.S.Manjunath, Detecting Packed Executables based on Raw Binary Data, VRL Technical Report, Jun. 2010

L. Nataraj, A. Sarkar and B. S. Manjunath, Improving re-sampling detection by adding noise, Proceedings of SPIE, 2010, Media Forensics and Security, vol. 7541, pp. 75410I-75410I-11, Jan. 2010

L. Nataraj, A. Sarkar and B. S. Manjunath, Adding Gaussian Noise to Denoise JPEG for detecting Image Resizing, International Conference on Image Processing, 2009, pp. 1493-1496, Cairo, Egypt, Nov. 2009

Anindya Sarkar, Lakshmanan Nataraj and B. S. Manjunath, Robust Key-point based Data Hiding, VRL Technical Report, Sep. 2009

Anindya Sarkar, Lakshmanan Nataraj and B. S. Manjunath, De-

tection of Seam Carving and Localization of Seam Insertions in Digital Images, 11th ACM Workshop on Multimedia and Security, pp. 107-116, Princeton, New Jersey, Sep. 2009

Anindya Sarkar, Lakshmanan Nataraj, B. S. Manjunath, Upamanyu Madhow, Estimation of Optimum Coding Redundancy and Frequency Domain Analysis of Attacks for YASS - a Randomized Block Based Hiding Scheme, Proc. IEEE International Conference on Image Processing, pp. 1292-1295, San Diego, CA, Oct. 2008

Abstract

A Signal Processing Approach To Malware Analysis

by

Lakshmanan Nataraj

There is an alarming increase in the amount of malware that is generated today. Several studies have shown that most of these new malware are just variants of existing ones. In this research we focus on developing orthogonal methods motivated by Signal and Image Processing. We exploit the fact that most malware variants are similar in structure. One could then treat malware as digital signals and apply Signal and Image Processing techniques to compute descriptions that facilitate detection and classification of malware. First, we will present SARVAM: Search And RetrieVAl of Malware, an online malware search and retrieval system where one can upload a binary executable and search over a database of approximately 7 million malware samples using Image Similarity metrics. Next, we generalize this approach by expanding malware as a sparse linear combination of other malware samples. Finally, the methods can be generalized to data forensics, where given a block of data we can determine the data type.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Sarvam Sattva Mayam

Everywhere there is purity

The number of malicious attacks have risen in recent times. Today's defense mechanisms are based on scanning systems for suspicious or malicious activity. If such an activity is found, the files under suspect are either quarantined or the vulnerable system is patched with an update. These scanning methods are based on a variety of techniques such as static analysis, dynamic analysis and other heuristics based techniques, which are often slow to react to new attacks and threats. Static analysis is based on analyzing an executable without executing it, while dynamic analysis executes the binary and studies its behavioral characteristics. Hackers are familiar with these standard methods and come up with ways to evade the current defense mechanisms. The traditional analysis techniques are known to hackers who produce new malware variants that easily evade these detection methods. They create a large number of malware vari-

ants from existing malware using inexpensive easily available "factory toolkits" in a "virtual factory" like setting, which then spread over and infect more systems. Once a system is compromised, it either quickly looses control and/or the infection spreads to other networked systems. While security techniques constantly evolve to keep up with new attacks, hackers too change their ways and continue to evade defense mechanisms. As this never-ending billion dollar "cat and mouse game" continues, it may be useful to look at avenues that can bring in novel alternative and/or orthogonal defense approaches to counter the ongoing threats. The hope is to catch these new attacks using orthogonal and complementary methods which may not be well known to hackers, thus making it more difficult and/or expensive for them to evade all detection schemes. *This dissertation focuses on such Orthogonal Solutions from Signal and Image Processing that complement standard approaches.*

1.1 Challenges

While most malware are geared towards Windows Operating System, they are also quickly expanding to other avenues such as Android, Linux and OS X. Antivirus vendor G-DATA reported that they discovered more than 1.5 Million malicious Android apps in 2014 and more than 400,000 apps in just the first quarter of 2015. Similarly, there has also been a stark rise in Linux malware and OS X malware. An important question in this context is: *Can we have a single method that can detect malware irrespective of which Operating System it comes from without having to know the nuances of each system?*

Traditional methods to analyze malware are using static code analysis methods

which usually disassemble an executable to study its control flow, and dynamic analysis techniques which execute the binary in a virtual environment to study its behavioral characteristics. However, a common way to defeat static analysis is by using packers on a executable which compress and/or encrypt the executable code and create a new packed executable that mimics the previous executable in function but reveals the actual code only upon execution runtime. Dynamic analysis is agnostic to packing but is slow and time consuming. Further, today's malware are designed to be Virtual Machine (VM) aware which either do not do any malicious activity in the presence of VM or attempts a "suicide" when a VM is detected. The challenges here are: *Can we design techniques that are fast, do not need disassembly, unpacking or execution?*

Finally, malware of today are constantly evolving and show up in forms other than an executable such as compressed malware, in PDF files and other documents. This poses an important question: *Can we identify malware when they are compressed or in other forms?*

A key emphasis in all the above-mentioned challenges is development of complementary methods that address the limitations of existing approaches. Alternative representations of malware data such as signals or images have patterns that are not captured by standard methods. We explore these types of representations in this dissertation.

1.2 Summary of Contributions

1. Image Similarity based Malware Analysis: A common method of viewing and editing malware binaries is by using *Hex Editors*, which display the bytes of the binaries in hexadecimal representation.An equivalent representation is viewing a binary as a

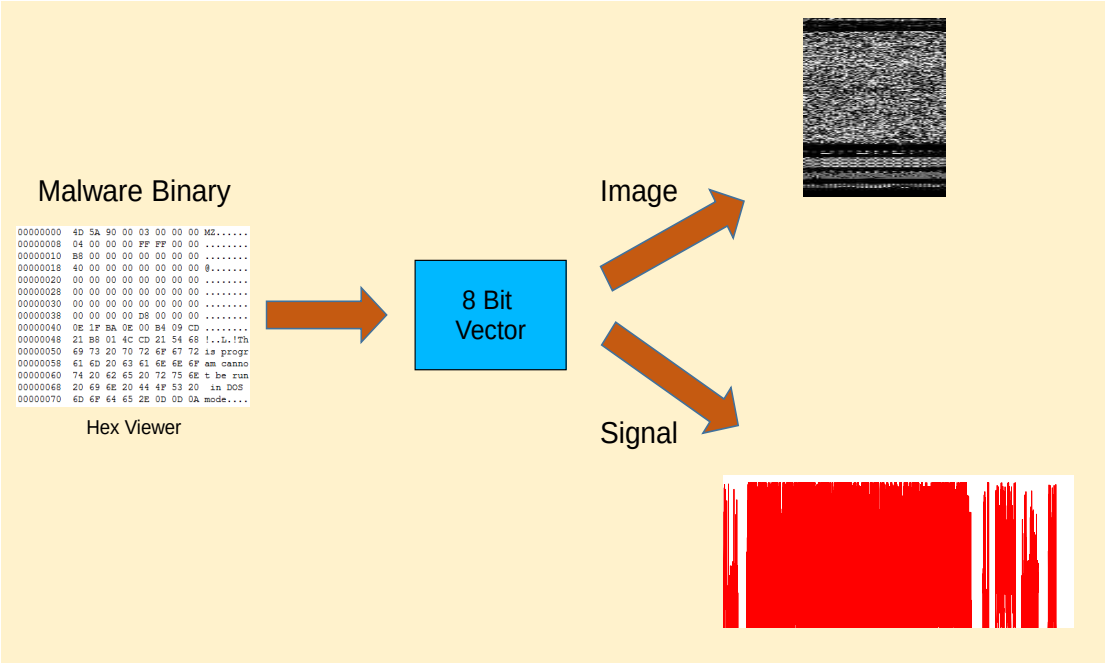


Figure 1.1: Malware represented as a Signal and an Image

grayscale image or a signal as shown in Fig. 1.1.

In the first part of the dissertation we represent malware binaries as digital grayscale images with the observation that malware variants that are similar in structure and from the same family also appear similar visually (Fig. 1.2). We then apply image descriptors to model the similarity between malware variants, identify malware families, separate malware from benign software and retreive similar malware from a large database. The advantages of our proposed method are:

- **Fast**: Since our method works on raw bytes, there is no need for disassembly or execution, thus making it faster than both static analysis and dynamic analysis.
- **Agnostic to Packing**: When unpacked malware variants from the same family are packed, the structure of the packed malware variants is different but there

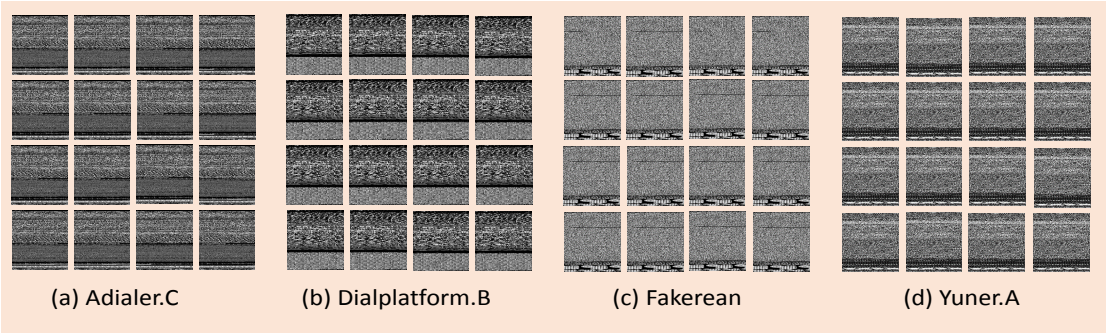


Figure 1.2: Visual similarity among malware variants of 4 different families

is still visual similarity among the packed malware variants, thus making our method agnostic to packing.

- **Agnostic to Operating System**: Since malware authors use similar techniques while creating malware variants irrespective of the Operating System, our technique works on malware from various Operating Systems such as Windows, Linux, Android and OS X.

2. Signal and Sparsity based Malware Analysis: In the second part we generalize the 2D image based representation further by representing malware as digtial signals. Motivated by the observation that variants from the same malware family have small changes, we model the variant as a sparse linear combination of other variants from the same family, and thus determine families of unknown malware . We further show that lower dimensional projections can also capture this similarity (Fig. 1.3).

3. Extensions to Data Type Classification: In the final part we extend the previous approaches to find similarities in compressed malware and identify their families. We further extend this to data forensics where the objective is to determine the type of a file, given a block of data.

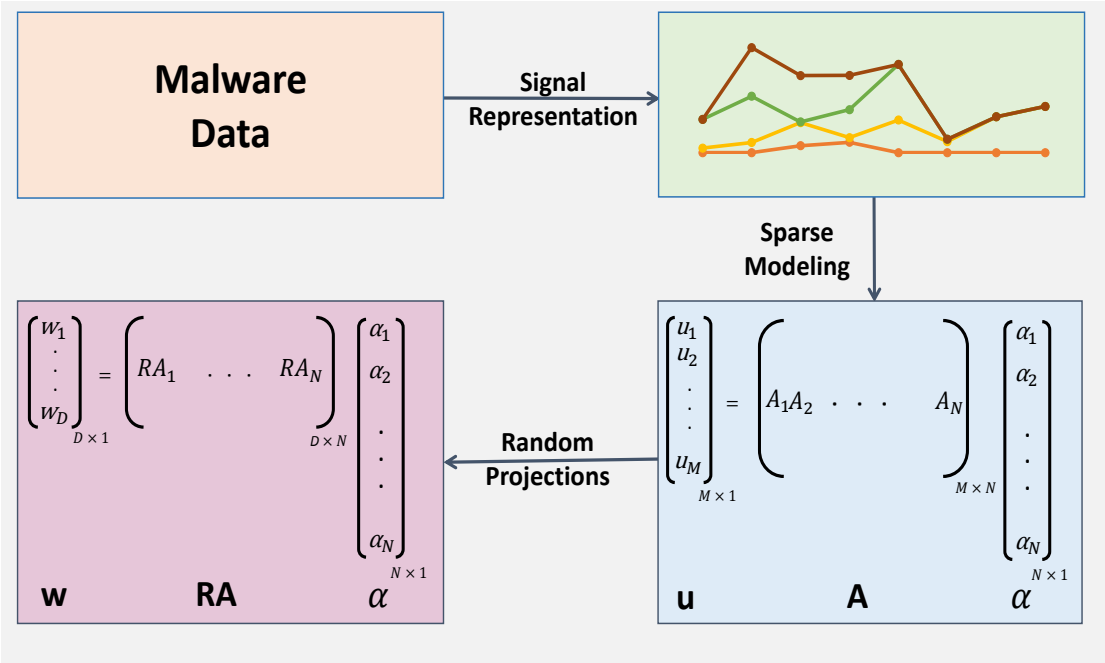


Figure 1.3: Malware samples are represented as numerical vectors, projected to lower dimensions and then modeled using the Sparse Representation based Classification (SRC) framework

1.3 Organization

The dissertation is organized as follows:

- In Chapter 3 we treat malware binaries as 2D signals (digital images) and bring in well known image processing methods to process, organize and search through such image based descriptions. With the observation that malware variants from the same families appear visually similar, we analyze similarity among malware by computing the Intra-family distances and Inter-family distances using global image descriptors. We then model malware family identification as a multi class classification problem using a Nearest Neighbor Classifier.

- In Chapter 4 we describe the systems we developed for malware detection and retrieval. In order to separate malware from benign software, we compute image descriptors on the whole executable as well as individual sections to obtain a richer descriptor, which we use for malware detection. Further, we designed an online system, SARVAM, to retrieve visually similar malware from a large database of malware.

- In Chapter 5 we generalize the 2D image based representation to a 1D signal based representation. We further model a malware variant belonging to a particular malware family as a linear combination of variants from that family. Since variants of a family have small changes in the overall structure and differ from variants of other families, we show that Random Projections of malware in lower dimensions preserve this "similarity".

- In Chapter 6 we deal with similarity among compressed malware variants and use a two layer meta learning based approach to find similarities among data types and compressed malware variants.

- In Chapter 7 we present some future directions based on this dissertation and conclude the dissertation.

Chapter 2

Overview of Malware Analysis

Malware - malicious software, is any software that is designed to cause damage to a computer, server, network, mobile phones and more such devices. Based on their function, malware are classified into different *Types* such as Trojans, Backdoors, Virus, Worm, Spyware, Adware and more. Malware are also identified by which *Platform* they belong to, such as Windows, Linux, AndroidOS and others. Apart from *Types* and *Platforms*, malware are further classified into *Families* depending on their specific function. These *Families* in turn have many *Variants* which perform almost the same function. The entire malware landscape is shown in Fig. 2.1. According to the Computer Antivirus Research Organization (CARO) convention for naming malware, a malware is represented by: *Type:Platform/Family.Variant*.

There is a tremendous increase in the amount of malware being generated today. Antivirus software vendor Kaspersky recently reported that they process on average 325,000 samples per day. The main reason for such a deluge is *malware mutation: the process of creating new malware variants from existing ones*. Variants are created

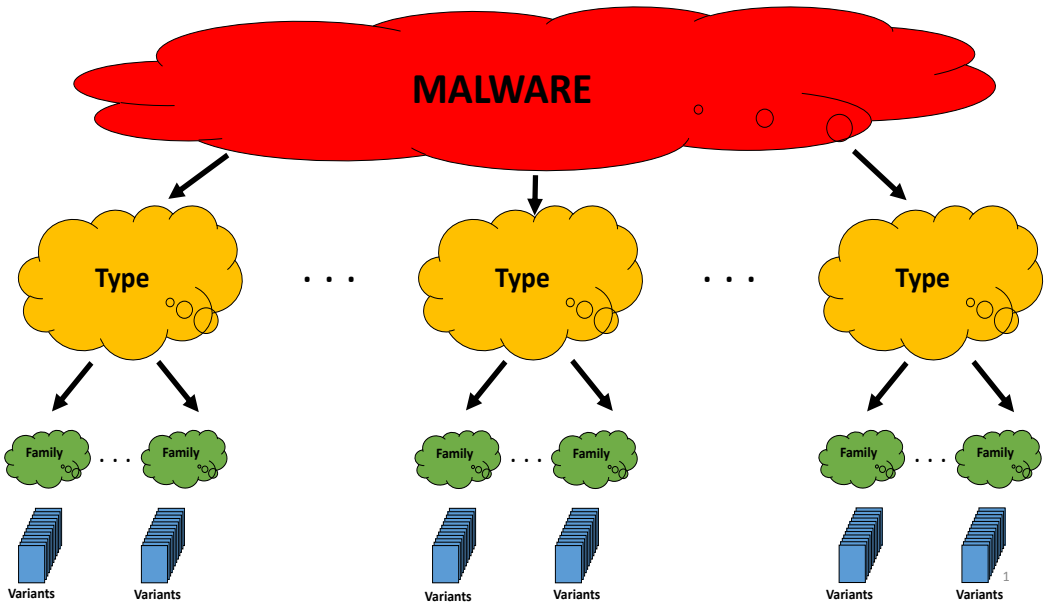


Figure 2.1: Malware Landscape

either by making changes to the malware code or by using executable packers. In the former case simple mutation occurs by changing small parts of the code. In the latter case a more complex mutation occurs either by compressing or encrypting (usually with different keys) the main body of the code and appending a decompression/decryption routine, which during runtime decompresses/decrypts the encrypted payload. The new variants perform the same function as the original malware but their attributes would be so different that Antivirus software, which use traditional signature based detection, would not work on them. Based on their function, these variants are classified into different *malware families*. As an example, if we assume there are M structurally unique malware samples belonging to M malware families, these M malware undergo N code mutations and we obtain MN samples. These are then packed using P packers

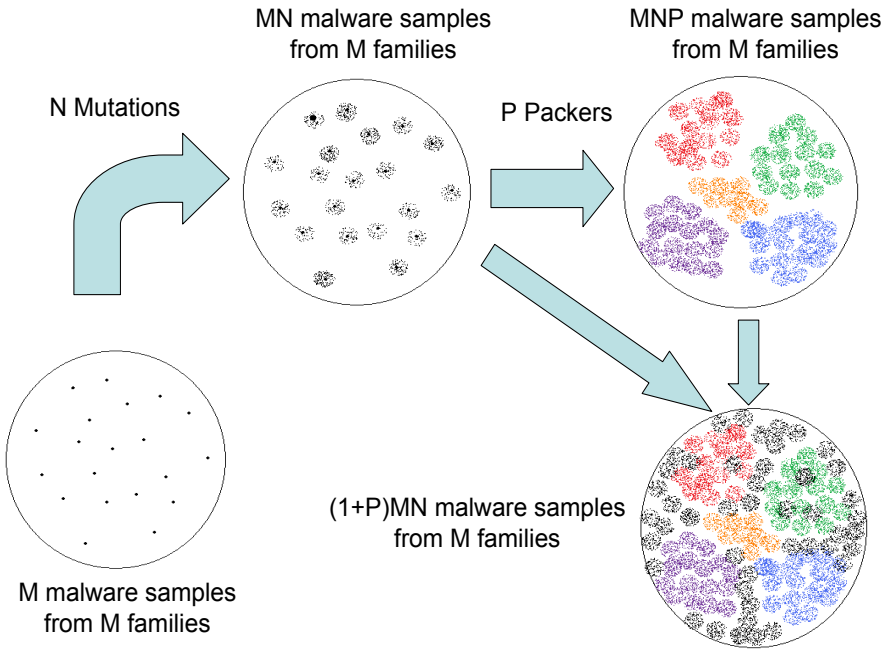


Figure 2.2: Example of Malware Evolution

to get MNP samples. The final dataset is a mixture of MNP packed samples and MN unpacked samples, a total of $MN(1+P)$ samples as shown in Fig. 2.2. In reality, apart from these samples there may be mixture of unpacked and packed malware from other families.

Malware classification deals with identifying the family of an unknown malware variant from a malware dataset that is divided into many families (Fig. 2.3). The level of risk of a particular malware is determined by what function it does which is in turn reflected in its family. Hence, identifying the malware family of an unknown malware is crucial in understanding and stopping new malware. It is usually assumed that an unknown malware variant belongs to a known set of malware families (supervised classification). Having a high *classification accuracy* (the number of correctly classified families) is desirable. A closely related problem is search and retrieval where the ob-

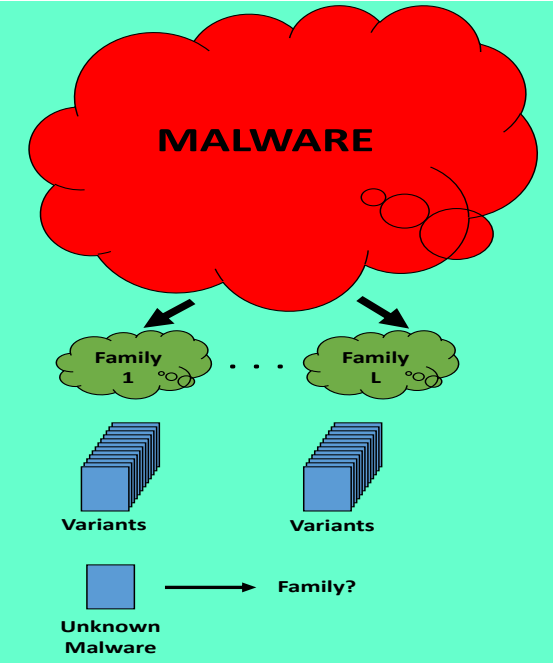


Figure 2.3: Illustration of Malware Classification

jective is to retrieve similar malware matches for a given query from a large database of malware.

In malware detection the problem is to determine if an unknown executable is malicious, benign or unknown as shown in Fig. 2.4. This problem is more challenging than malware classification where all samples are known to be malicious. High *detection accuracy* (the number of correctly classified malware) and low *false positives* (number of benign misclassified as malware) are important. For example, if an essential benign system file is falsely classified as malicious, then it can result in system not working properly.

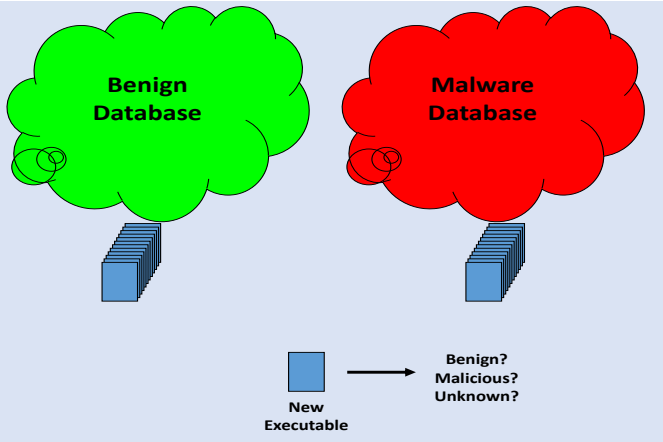


Figure 2.4: Illustration of Malware Detection

2.1 Malware Analysis

Malware analysis can be broadly classified into *Static Analysis*, *Dynamic Analysis* and *Statistical Analysis* (Fig. 2.5). We will briefly describe each below.

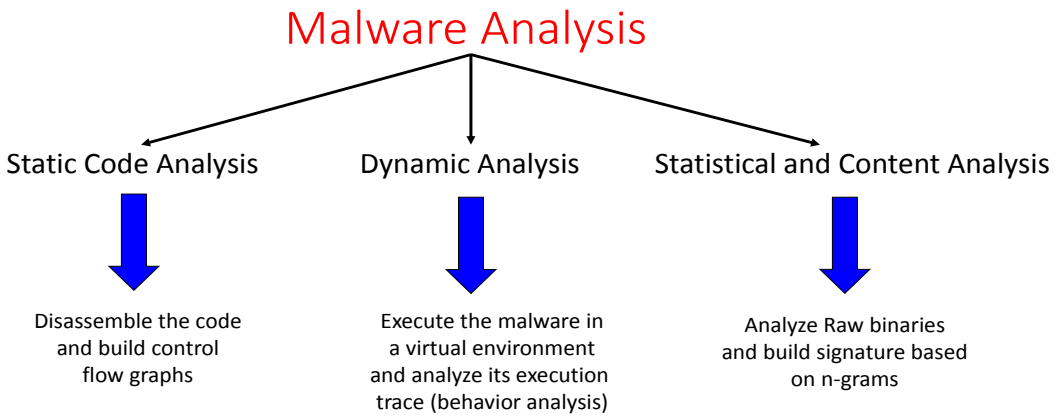


Figure 2.5: Types of Malware Analysis

2.1.1 Static Analysis

As the name suggests, static analysis is based on analyzing a malware without executing it. These techniques study the functioning of an executable by disassembling the executable and then extracting features. The most common static code based analysis method is control flow graph (CFG) analysis. After disassembly, the control flow of the malware is obtained from the sequence of instructions and graphs are constructed to uniquely characterize the malware. The control flow graphs are decomposed into a set of subgraphs of fixed size k. A graph similarity measure is used to measure the similarity between two malware. The similarity measure between two samples is calculated as follows:

$$CFG\ similarity = \frac{number\ of\ matching\ subgraphs}{total\ number\ of\ subgraphs}$$

.

The control flow graphs of known malware are stored in a database. For an unknown executable, the control flow graph is extracted and compared with the database to find a match. Although this approach is promising, it does not work on packed/obfuscated malware since the control flow of a packed malware reveals only the unpacking routine and not the actual flow.

2.1.2 Dynamic Analysis

In dynamic analysis the behavior of the malware is traced by running the malware executable in a virtual sandboxed environment for several minutes. The most commonly used approach is system-call level monitoring. A sequential report of the moni-

tored behavior for each malware binary is generated based on the performed operations and actions. The report typically includes all system calls and their arguments are stored in a representation specifically tailored to behavior-based analysis. Another approach called forensic snapshot comparison relies on a combination of features that are based on comparing the pre-infection and post-infection system snapshots. A key difference between system-call monitoring and forensic comparison, is that the latter approach does not capture the temporal ordering of forensic events. While dynamic analysis is promising, it is time consuming since the malware has to be observed for several minutes. Also, some of the most recent malware are designed to be "Virtual Machine" aware and do not perform malicious activities if a virtual machine is detected.

2.1.3 Statistical and Content Analysis

Statistical and content analysis based techniques are based on a variety of techniques: n-grams, n-perms, hash based techniques and file structure based techniques. The most common of these are n-grams based techniques. The n-grams signature of a string is a set of all substrings of the string with a length n. In the case of a binary executable, the n-grams signature is usually computed on the string of its raw bytes, or disassembled instructions. Various similarity measures have been proposed to compare the similarity between n-grams signatures. The Pearsons χ^2-test has been proposed as a similarity measure, where the n-grams signature also includes the frequency distribution of individual n-grams. For a faster and scalable computation of similarity, the Jaccard similarity metric has been extensively used. Jaccard similarity is the number of common occurrence of n-grams in both n-grams signatures with respect to the total number of unique n-grams. More precisely, given two n-grams signatures (sets of

n-grams) s_a and s_b:

$$J(s_a, s_b) = \frac{s_a \cap s_b}{s_a \cup s_b}$$

. However, n-grams based approaches are less scalable because of the computationally expensive feature matching operation over relatively large dimensionality of the n-grams feature space.

File structure based techniques extract simple features from the file header such as length of the file, number of sections, length of sections and more. A compact feature vector based on the above entities is used to characterize a malware. Although the method is fast and simple, a malware can easily be designed to have a file structure that mimics a benign software.

2.2 Summary of Related Work

This chapter provided some insights on malware analysis. More information on control flow graph (CFG) analysis can be found in [36, 33, 61, 51, 53]. Karim et al. examine the problem of developing phylogenetic models for detecting malware that evolves through code permutations [54, 106]. Carrera et al. develop a taxonomy of malware using graph-based representation and comparison techniques for malware [36]. Hu et al. [51] proposed function call graphs to implement an efficient nearest-neighbor search on a large graph database of malware. Kruegel et al. [61] extracted CFGs from network streams and detected polymorphic worms by identifying structural similarities. Worm detection and classification occurs by identifying the prevalence of k-subgraph features between worm-like executable content and unknown executable content.

For more on behavioral profile based dynamic analysis, see [63, 24, 56, 85, 27, 86]. Some works generate a human readable report of the execution flow and extract features from the reports [85, 86]. Bailey et al. were among the first to point out inconsistencies in labeling by popular AV vendors and present an automated classification system as a solution for robustly classifying malware binaries through offline behavioral analysis [24]. Another related work is from Kolbitsch et al. [56], where dynamic analysis is used to build models of malware programs based on information flow between system calls. Similarly, Bayer et al. [27] propose an unsupervised learning system to automatically cluster malware based on their behavioral profile. Although they obtained high precision and recall on a small dataset, Li et al. [64] argued that the high precision and recall of the above technique is due to selection bias in generating their ground truth. In [79], Park et al. classified malware based on detecting the maximal common subgraph in a behavioral graph. Rieck et al. proposed a supervised learning approach to behavior based malware classification [85]. They use a labeled malware dataset comprising many families and monitored the behavior of the samples in a sandbox environment from which behavioral reports were generated. From every report, they generated feature vectors based on the frequency of some specific strings and used Support Vector Machines for classification. In [86], they extend this technique and build a system which combines both classification and clustering. Several tools exist for sandboxed execution of malware including TTanalyze [105], CWSandbox [108], Norman Sandbox [11], Anubis [1], Cuckoo Sandbox [3] and Malwr [9]. While CWSandbox and TTanalyze leverage API hooking, Norman Sandbox implements a simulated Windows environment for analyzing malware. A complementary analysis is proposed in [68], where a layered architecture is proposed for building abstract models based on run-

time system call monitoring of programs. Another approach based on forensic snapshot comparison is presented in [113].

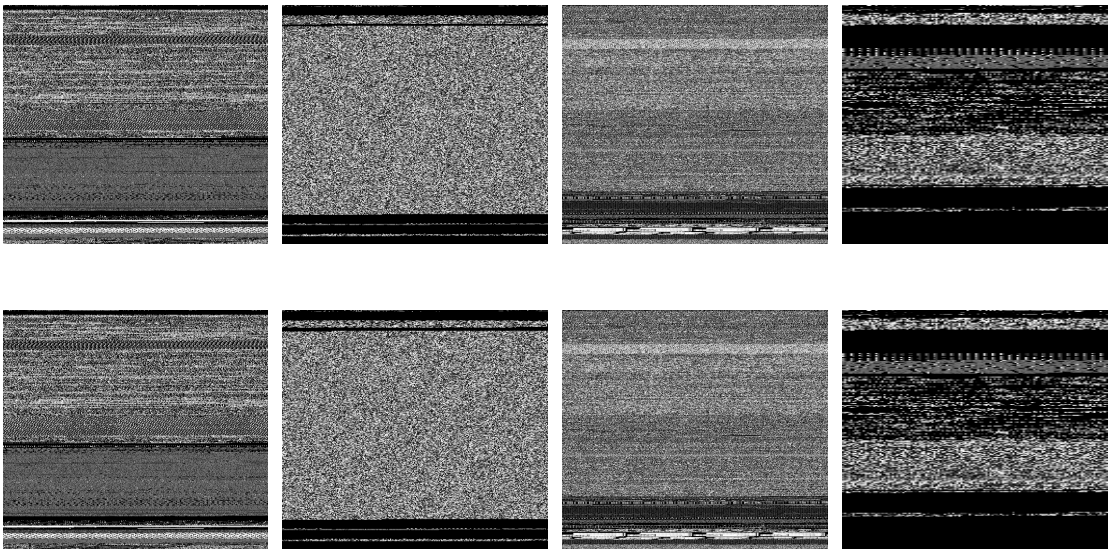
There are several approaches that use statistical and content based methods, the most common being n-grams analysis [21, 57, 58, 54, 81, 91, 53] and n-perms [54, 52, 62]. Kolter and Maloof [57] were among the first to use n-grams of raw binary data for malware detection. A malware phylogeny generation technique was proposed using n-perms to match every possible permuted code [54]. Jacob et al. [52] studied the preserved statistical similarity over packed binaries, and proposed a packer-agnostic bigrams based similarity measure. Jang et al. [53] proposed feature hashing to reduce the high-dimensional feature space in malware analysis, and implemented feature hashing on n-grams based features. However, its evaluation was performed on the clustering of an unpacked malware dataset, and no benign samples were used to test the accuracy of the system. Apart from n-grams, there are also techniques based on file structure statistics [93, 95, 84] and computing hashes on malware [60, 107]. Schultz et al. [93] proposed the extraction of static features from the header and the resource section of executables for the detection of malware. Shaq et al. [95] proposed PE-Miner, a similar approach, based on 189 features from the structural information of the PE file. In another work, only seven PE based features were used to produce comparable detection results [84]. Among hash based methods, *ssdeep* [60] is a common technique to compute context triggered piecewise hashes on raw binaries. *Pehash* [107], however, uses the Portable Executable (PE) file structure to compute a hash.

Chapter 3

Malware Images

3.1 Introduction

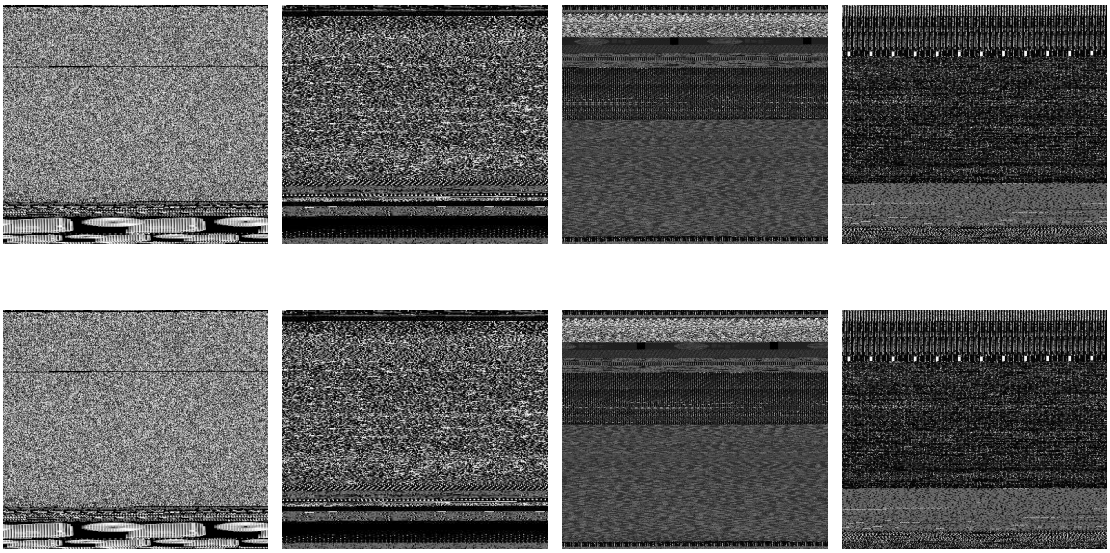
We propose a novel method to analyze malware by representing malware binaries as digital signals and images. Based on the observation that variants belonging to the same malware family exhibit structural and visual similarity, we use Image similarity techniques to capture the similarity between the variants. Examples of such variants are shown in Fig. 3.1. More specifically, we combine some of the most commonly used image similarity features in one setting and define them as *Generalized Features*. This allows us to not depend on one specific feature but rather use a suite of methods which can be tuned according to the problem at hand. As opposed to current malware analysis techniques, our approach does not require disassembly, decompiling, deobfuscation or execution of the binary. This allows us to capture patterns that are not usually captured by standard techniques. Using the large number of malware samples available in the wild, we employ supervised learning based techniques to model *malware*



(a) Adialer (b) Alueron (c) Autorun (d) Dontovo

Figure 3.1: Examples of malware variants from various Windows malware families. The top row shows grayscale visualizations of malware from four families: (a) Adialer, (b) Alueron, (c) Autorun and (d) Dontovo. The bottom row are the variants from the corresponding families. We observe two main points: (1) there is similarity in variants within families, (2) they can be distinguished from variants of other families.

similarity and distinguish them between different families-*malware classification*. Experimental results on various datasets show that our approach is better than most static analysis based methods and comparable to dynamic analysis based methods. Another advantage of our approach is that it is independent of the Operating System (OS) for which a malware is designed for. Today's malware are no longer restricted to Windows platforms and are rapidly expanding to other platforms such as Android [4], Linux [6] and MAC OS X [17]. The rise has been stark in the case of Android and Anti Virus



(a) Win (b) OS X (c) Linux (d) Android

Figure 3.2: Examples of malware variants from different OS: (a) Win 32 variants of Fakerean family, (b) MAC OS X variants of Wirenet family, (d) ELF malware which are Backdoors, (d) Android variants of Kungfu1. These clearly show that the visual and structural similarity hold for malware variants irrespective of their platform

software vendor G Data reported that they discovered 1,548,129 new Android malware samples in 2014 [4]. However, the malware variants, irrespective of their platforms, still have similar structure since the underlying techniques in creating variants more or less remains the same. This is further illustrated in Fig. 3.2, where we clearly see that variants of different platforms still exhibit similarity. While static and dynamic analysis methods have to be re-developed for various OS platforms and their versions, our method would at most need tuning of parameters. Experiments on malware datasets from various OS further confirm this (refer Sec. 3.4.4 and Sec. 3.5.2).

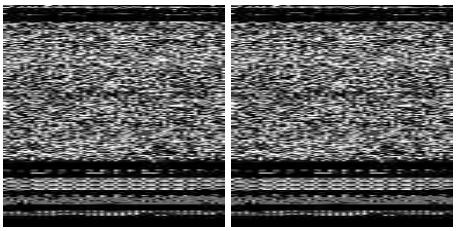
The rest of this chapter is organized as follows. Sec. 3.2 describes the image representation of malware binaries and how malware variants appear when visualized as digital images. Sec. 3.3 combines some of the common image similarity descriptors in one setting called Generalized Features and explains the steps while computing the feature. Similarity Analysis of malware variants is presented in Sec. 3.4. Sec. 3.5 explains classification of malware variants into different families with detailed experiments. The summary is presented in Sec. 3.6.

3.2 Visualizing Unpacked and Packed Malware

A common method of viewing and editing malware binaries is by using *Hex Editors*, which display the bytes of the binaries in hexadecimal representation.An equivalent representation is viewing a binary as a grayscale image or a signal.The range of this signal is $[0, 255]$ (0: black, 255: white). In the case of an image, the width of the image is fixed and the height is allowed to vary depending on the file size. Examples of these images are shown in Fig. 3.1 and Fig. 3.2.

Most **unpacked** malware variants are created using simple mutation techniques. Fig. 3.3 shows an example of unpacked malware variants of the recently exposed Regin malware [16]. Since simple mutation techniques are used, these variants differ in only 7 bytes.

An executable can be **packed** using methods such as compression, encryption or a combination of both. Packers transform the original executable's binaries to a different form and generate new executables by embedding the compressed, possibly encrypted code with an appended loading routine. This routine is responsible for the automated

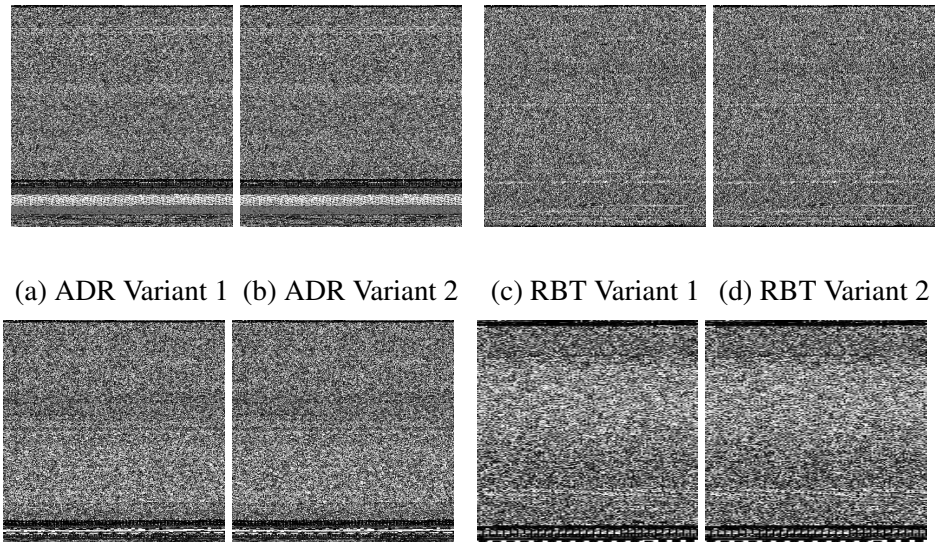


(a) Regin Variant 1 (b) Regin Variant 2

Figure 3.3: In this example, we see visualizations of unpacked variants ((a),(b)) of the recently exposed Regin malware [16]. Both these variants are of the same length (13,284 bytes) but only 7 bytes (0.0527 %) are different.

decompression and decryption of the code before executing them. There are hundreds of packers that exist today, both commercial and open source. This makes it very easy for malware writers to create new malware variants which are not detectable by traditional anti-virus software. Hence, these software resort to either creating a new signature for every new packed threat it encounters or try to emulate the executable code and then scan the image of the code in memory. In general, it is believed that nearly 80% of malware are packed [66, 49] and 50% of existing malware are packed versions of old malware [96]. Even detecting if an executable is packed or not is a challenging problem [66, 80, 44, 73].

When two unpacked variants belonging to a specific malware family are using a packer to obtain packed variants of the same family, their structure no longer remains the same as that of the unpacked variants. However, the structure within the packed variants are still similar though the actual bytes may vary due to compression and/or encryption. Fig. 3.4 shows examples of packed malware variants of two different families (Adialer and Azero) that are packed using UPX packer. We observe that within



(a) ADR Variant 1 (b) ADR Variant 2 (c) RBT Variant 1 (d) RBT Variant 2

(e) ADB Variant 1 (f) ADB Variant 2 (g) AZO Variant 1 (h) AZO Variant 2

Figure 3.4: Examples of UPX packed malware variants: (a) Adialer (ADR), (b) Rbot (RBT), (d) Adultbrowser (ADB), (d) Azero (AZO). We see that even among packed variants, visual similarity is maintained among families.

families, the variants exhibit structural similarity and is absent across families. Since there is visual and structural similarity, for both unpacked and packed malware variants, we will use global image similarity descriptors and obtain compact signatures for these variants, which are then used for similarity and classification as explained in the next sections.

3.3 Generalized Features for Malware Similarity

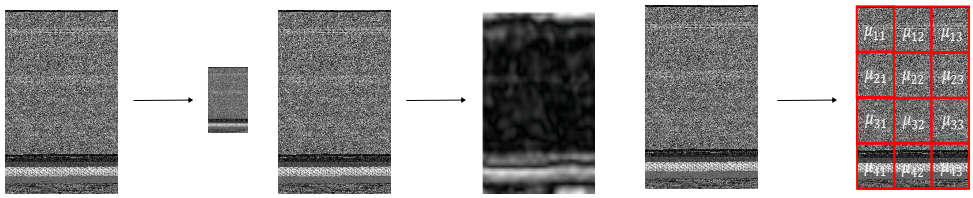
We define Generalized Features (GF) as compact signatures that are computed on an image using a generic set of parameters. These are common parameters such as number of filters used, size of blocks that an image is divided into and statistical quantities

such as mean and standard deviation. Global image similarity features such as Color Layout Descriptor (CLD) [90], Homogeneous Texture Descriptor (HTD) [67, 90] and GIST [77, 102] use a combination of these parameters. These descriptors are explained later in Sec. 3.3.2.

Let $\mathbf{X_m} \in \mathbb{R}^L$ represent a malware sample, where L is the length (number of bytes) of the malware. For simplicity, we assume that L is an even number in order to represent the malware as a digital grayscale image $\mathbf{X_g}$ of dimensions $L_w \times L_h$, where L_w,L_h represent the width and height of the image (if L is odd, it can be made even by zero padding). Let $s_{ov} \in Z^+$ be the parameter that divides an image into overlapping or non-overlapping blocks ($s_{ov} \geq 0$, 0 being no overlap), (b_x, b_y) be the block sizes that an image is divided into in the horizontal and vertical directions. For practical purpose, an image is resized to a standard size and this can be parameterized as $\mathbf{R_s} = (R_{sx}, R_{sy})$, where R_{sx}, R_{sy} are the resizing factors in the horizontal and vertical directions.. Let N_f be the number of filters used to filter the image. This can be a single filter or a bank of filters (such as Wavelets or Gabor filters). Finally, let $\mathbf{S_q}$ denote the set of statistical quantities used while computing the feature. $\mathbf{S_q}$ usually includes mean, standard deviation and other statistical quantities. We define Generalized Feature $\mathbf{G_F}$ as follows:

$$\mathbf{G_F} = f(\mathbf{X_g}, [\mathbf{R_s}, (b_x, b_y), s_{ov}, N_f, \{\mathbf{S_q}\}]) \tag{3.1}$$

where $f(\cdot)$ is a function that takes the malware image $\mathbf{X_g}$ as input and generates a feature vector based on various parameters.



(a) Resizing (b) Filtering (c) Block Averaging

Figure 3.5: Illustration of some of the common operations that are carried out while computing image similarity descriptors.

3.3.1 Generalized Features as Linear Measurements

Some of the common operations that are performed while computing image similarity features are resizing, filtering and block averaging. These operations can be represented as obtaining linear measurements on an image.

Resizing: Resizing an image to a standard size is a common practice while computing image similarity features for speed and compactness [43, 101]. The malware image $\mathbf{X_g}$ is resized (usually downsampled) to a smaller image $\mathbf{X_r}$ (usually square) of length L_r. Let $\mathbf{R_s}$ be resizing factor and $\mathbf{R_M}$ be the $L_r \times L$ resizing matrix (we assume linear interpolation). The resizing operation can be expressed as taking linear measurements on an image,

$$\mathbf{X_r} = \mathbf{R_M X_g} \tag{3.2}$$

For example, if the image is downsampled by a factor of 2 ($\mathbf{R_s} = (0.5, 0.5)$), then

$$\mathbf{R_M} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{3.3}$$

Filtering: Filtering is another operation that is usually carried out while computing image similarity features. An image is passed through a series of filters N_f and statistical quantities are computed from the filtered images to obtain a rich image signature. For example, when computing HTD or GIST, oriented Gabor features are computed on an image at different scales (refer Sec. 3.3.2).

Block Averaging: Block averaging is the process of dividing an image into smaller blocks and computing the average intensity value on those blocks. Similarity descriptors such as Color Layout Descriptor (CLD) and GIST use block averaging. Here, we show how the block averaging operation can be expressed in terms of linear measurements. Consider a $L_W \times L_H$ malware image $\mathbf{X_g}$ ($L = L_W L_H$). This image is divided into smaller $Q_W \times Q_H$ blocks ($Q = Q_W Q_H$) on which the average value is computed to form a $Q_W \times Q_H$ feature matrix $\mathbf{h}$. The block averaging can be represented as taking inner products of the image with a set of Q masks $\{\boldsymbol{\phi}_1, \boldsymbol{\phi}_2, ..., \boldsymbol{\phi}_Q\}$, each of dimensions $L_W \times L_H$ which have values for pixel positions of the corresponding block and zero elsewhere. Let $\{h_1, h_2, ..., h_Q\}$ be the values of $\mathbf{h}$. Then $h_1 = \langle \mathbf{X_g} \cdot \boldsymbol{\phi}_1 \rangle$, $h_2 = \langle \mathbf{X_g} \cdot \boldsymbol{\phi}_2 \rangle, ..., h_Q = \langle \mathbf{X_g} \cdot \boldsymbol{\phi}_Q \rangle$, where $\langle \cdot \rangle$ represents the inner product operation. All these operations can be represented as a linear system of equations:

$$\underline{\mathbf{h}} = \mathbf{\Phi}\ \underline{\mathbf{X}}_{\mathbf{g}} \tag{3.4}$$

where $\underline{\mathbf{h}} = [h_1, h_2, ..., h_Q]^T$ is a $Q \times 1$ vector formed by vectorizing $\mathbf{h}$, $\underline{\mathbf{X}}_{\mathbf{g}}$ is a $L \times 1$ vector formed by vectorizing the image $\mathbf{X}_{\mathbf{g}}$ and $\mathbf{\Phi}$ is the $Q \times L$ fat transformation matrix, $(L >> Q)$. The rows of $\mathbf{\Phi}$ are formed by stacking the set of Q masks in vectorized form, $\mathbf{\Phi} = [\underline{\boldsymbol{\phi}}_1^T; \underline{\boldsymbol{\phi}}_2^T; ...; \underline{\boldsymbol{\phi}}_Q^T]$, where $\underline{\boldsymbol{\phi}}_i$, $(i = 1, 2, ..., Q)$ are the $L \times 1$ vectorized form of masks $\boldsymbol{\phi}_i$.

The dimensionality of $\mathbf{G_F}$ depends on the number of filters N_f, number of blocks N_b and the number of statistical quantities N_{S_q}. If no filtering is done, then N_f is assumed to be 1. The dimensionality is given by $d_{\mathbf{G_F}} = N_f N_b N_{\mathbf{S_q}}$ feature descriptor.

3.3.2 Common Similarity Descriptors

Color Layout Descriptor (CLD): This descriptor [90] divides an image into smaller blocks which are factors of 8×8. The average value of every smaller block is computed to obtain a 8×8 matrix. The Discrete Cosine Transform (DCT) is computed on this matrix and the top coefficients form the descriptor. We can express CLD in terms of Generalized Features as shown in (3.5). In this case, $S_q = \mu, (b_x, b_y) = (8, 8), \mathbf{R_s} = (1, 1), N_f = 1, s_{ov} = 1$. The dimensionality is 64 ($N_f = 1, N_b = 64, N_{\mathbf{S_q}} = 1$).

$$\mathbf{G_F} = f(\mathbf{X_m}, [(1, 1), (8, 8), 0, 1, \mu]) \tag{3.5}$$

Homogeneous Texture Descriptor: One of the common descriptors to classify texture images is the Homogeneous Texture Descriptor (HTD) [67, 90]. An image is passed through a set of Gabor filters of various scales and orientations to obtain a set of filtered images. The set of means and the standard deviations of all filtered images form the descriptor. In this case, $S_q = \{\mu, \sigma\}, (b_x, b_y) = (1, 1), (Rx, Ry) = (1, 1), s_{ov} = 0, N_f = 24$ and the dimensionality is 48 ($N_f = 24, N_b = 1, N_{\mathbf{S_q}} = 2$). The Generalized

Features representation of HTD is given in (3.6),

$$\mathbf{G_F} = f(\mathbf{X_m}, [(1,1),(1,1),0,24,\{\mu,\sigma\}]) \tag{3.6}$$

GIST Descriptor: This descriptor [77, 102] is a popular global image descriptor that describes the texture and spatial layout of a scene. An image is filtered with filters of various scales and orientations and then divided into blocks. The set of average values on all blocks form the descriptor. For GIST, $S_q = \mu, (b_x, b_y) = (4,4), (Rx, Ry) = (1,1), N_f = 32$ and dimensionality is 512 ($N_f = 32, N_b = 16, N_{\mathbf{S_q}} = 1$). However, in some works [43], the dimensionality is 320, when only $N_f = 20$ filters are used. The Generalized Features representation of GIST descriptor is given in (3.7),

$$\mathbf{G_F} = f(\mathbf{X_m}, [(1,1),(4,4),0,32,\mu]) \tag{3.7}$$

We will use GIST descriptors in our experiments since it has a richer description and also gives better performance in terms of accuracy (Sec. 3.5.3).

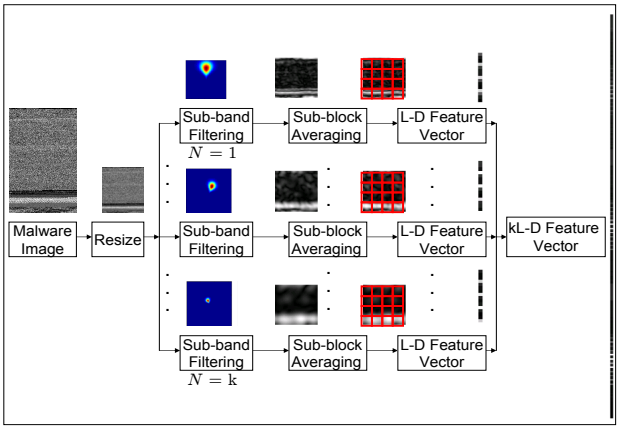


Figure 3.6: Steps to compute GIST descriptor on a malware

3.4 Malware Similarity

The objective here is to determine how much a malware variant is similar to another variant of the same family. We measure the similarity between malware variants in terms of Intra-Family distances and Inter-Family distances.

Consider a dataset of N labeled malware belonging to L different malware families with P malware per family. Let $\mathbf{G_F}^i_k$ represent the feature descriptor of the i-th family and k-th sample in that family ($i = 1, 2, ..., L$ and $k = 1, 2, ..., P$). The features are arranged such that the first P features are from family 1, the next P from family 2 and so on. Let $\mathbf{D_M}$ be the $N \times N$ distance matrix obtained by computing pairwise Euclidean distance on all the features. Let $\{\mathbf{G_F}^i_1, ..., \mathbf{G_F}^i_k, ...\mathbf{G_F}^i_P\}$ represent the features of the i-th family. The entries of $\mathbf{D_M}$ are distances $\{d^{i,j}_{k,l}\}$, where the indices i, j keep track of the family and k, l keep track of the individual samples both within and across families. We define a $P \times P$ matrix, $\mathbf{S}^{ij}$, whose entries are distances $\{d^{i,j}_{k,l}\}$ computed between families i and j, where $k = 1, 2, ..., P$ and $l = 1, 2, ..., P$. Further for every family i, we define a $P \times N$ matrix $\mathbf{D}^i$ formed by concatenating $\mathbf{S}^{ij}$ such that:

$$\mathbf{D}^i = [\mathbf{S}^{i1}, \mathbf{S}^{i2}, ..., \mathbf{S}^{iL}], \;\; i = 1, 2, ..., L \tag{3.8}$$

The complete distance matrix $\mathbf{D_M}$ is now formed by stacking the block matrices $\mathbf{D}^i$:

$$\mathbf{D_M} = [\mathbf{D}^1; \mathbf{D}^2; ...; \mathbf{D}^L] \;\in\; \mathbb{R}^{N \times N} \tag{3.9}$$

3.4.1 Intra-Family Distances

These are pairwise distances computed on features belonging a family and used to measure the amount of variation within that family. The set of features belonging to

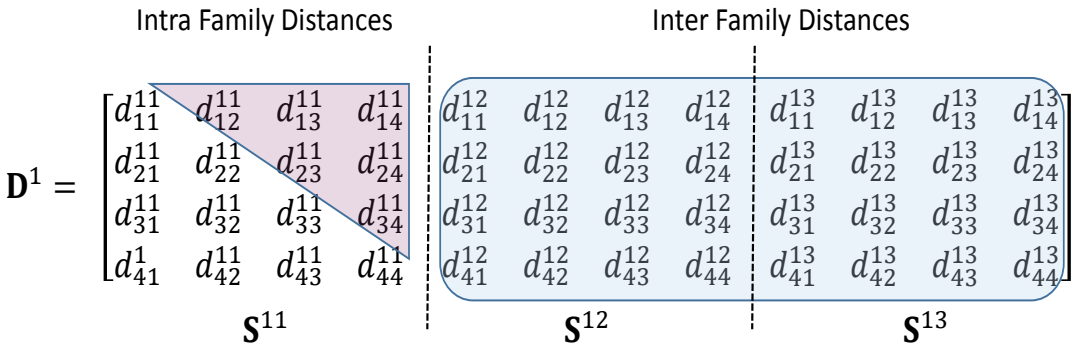


Figure 3.7: A simple example to illustrate Intra-Family Distances and Inter-Family Distances. We consider a dataset with $N = 12$ samples, $L = 3$ families and $P = 4$ samples per family. If we consider a particular family $i = 1$, we get a 4×12 matrix $\mathbf{D}^1 = [\mathbf{S}^{11}, \mathbf{S}^{12}, \mathbf{S}^{13}]$. The upper triangular part of $\mathbf{S}^{11}$ (excluding the diagonals) are the Intra-Family Distances while the entries of $\mathbf{S}^{12}$ and $\mathbf{S}^{13}$ are the Inter-Family Distances for this family.

family i are given by $\{\mathbf{G_F}^i_1, ..., \mathbf{G_F}^i_k, ...\mathbf{G_F}^i_P\}$. Computing pairwise distances on these features will result in a $P \times P$ symmetric distance matrix whose entries are given by $\{d^{i,i}_{k,l}\}$ where $k = 1, 2, ..., P$ and $l = 1, 2, ..., P$. Let $\mathbf{d_{intra}}^i$ represent the set of Intra-Family Distances for family i. This results in $\frac{P(P-1)}{2}$ distances per family given by:

$$\mathbf{d_{intra}}^i = \{d^{i,i}_{k,l}\} \tag{3.10}$$

where $i = 1, 2, ..., L$, $l > k,\ l \neq k,\ k = 1, 2, ..., P$ and $l = 1, 2, ..., P$.

3.4.2 Inter-Family Distances

These are pairwise distances computed across families and measure the dissimilarity across different families. Let $\mathbf{d_{inter}}^i$ represent the set of Inter-Family Distances for

family i. This results in $P(N-P)$ distances per family given by:

$$\mathbf{d_{inter}}^{i} = \{d_{k,l}^{i,j}\} \tag{3.11}$$

where $i = 1, 2, ..., L$, $j \neq i$, $k = 1, 2, ..., P$ and $l = 1, 2, ..., P$.

We illustrate the computation of these distances with a toy example. Consider a dataset with $N = 12$ samples, $L = 3$ families with $P = 4$ samples per family. This will result in 6 Intra-Family Distances and 32 Inter-Family Distances per family as shown in Fig.3.7.

3.4.3 Measuring Similarity

For every family i where $i = 1, 2, ..., L$, we compute the Intra-Family Distances (Eqn. (3.10)) and Inter-Family Distances (Eqn. (3.11)). Next for every variant in a family, we compute the minimum value of the Intra and Inter-family distances within and across families. The rationale behind computing the minimum value is that as long as there are two variants in a family that are similar, their pairwise distance will be small. For example, for the first variant of family 1, this would correspond to computing the minimum value of the first row of $\mathbf{S}^{11}$ (excluding the distance with itself d_{11}^{11}). For every family i, the Intra-Family Minimum Distances ($\mathbf{d_{intra}^{min}}^{i}(k)$)are given by:

$$\mathbf{d_{intra}^{min}}^{i}(k) = \min_{l} \{\mathbf{S}^{ii}[k, l]\} \tag{3.12}$$

where $k = 1, 2, ..., P$, $l = 1, 2, ..., P$, $l \neq k$ and $i = 1, 2, ..., L$. At the same time, there should not be a variant similar in other families. For this we compute the Inter-Family Minimum Distances ($\mathbf{d_{inter}^{min}}^{i}(k)$) across families as:

$$\mathbf{d_{inter}^{min}}^{i}(k) = \min_{j,l} \{\mathbf{S}^{ij}[k, l]\} \tag{3.13}$$

where $k = 1, 2, ..., P,\ l = 1, 2, ..., P$ and $i = 1, 2, ..., L,\ j = 1, 2, ..., L,\ j \neq i$. For the first variant of family 1, this would correspond to finding the minimum value in the first rows of both $\mathbf{S}^{12}$ and $\mathbf{S}^{13}$. These would result in P distances per family. We then compute the median value of these Intra and Inter-Family Minimum Distances:

$$\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}(i) = \text{median } \{\mathbf{d}^{\mathbf{min}\ i}_{\mathbf{intra}}(k)\} \tag{3.14}$$

$$\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}(i) = \text{median } \{\mathbf{d}^{\mathbf{min}\ i}_{\mathbf{inter}}(k)\} \tag{3.15}$$

where $i = 1, 2, ..., L,\ k = 1, 2, ..., P$, $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}(i)$ and $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}(i)$ represent the median values of the Intra and Inter-Family Minimum Distances for family i. We will use these values to measure the similarity within a family and the dis-similarity across families.

3.4.4 Analysis on Unpacked Malware

We consider four datasets of unpacked malware from various Operating Systems.

Windows Malware Dataset: This dataset consists of Portable Executable (PE) malware variants from 10 different families: *Adialer.c*, *Adultbrowser*, *Azero.a*, *Casino*, *Dorfdo*, *Jhee.v*, *Magiccasino*, *Podnhua*, *Rbot.gen* and *Yuner.a*, with each family having 10 variants. These were obtained and pruned from various sources including [72, 86].

Linux Malware Dataset: This dataset contains ELF malware binaries that were obtained and pruned from [18]. The labels were obtained from [19] and divided into 8 different families, with each having 10 variants: *Tsunami*, *Agent.T*, *Matrics.A*, *Grip!gen0*, *OSF.A*, *RST.B*, *Telf.8000* and *Adore*.

Android Malware Dataset: This dataset was obtained and pruned from the Android Malware Genome Project [114]. To measure similarity, the Dalvik Executable (DEX) file from the Android Application Package (apk) was extracted and the dataset was di-

vided into 10 families with 10 variants each: *ADRD*, *AnserverBot*, *BaseBridge*, *DreamLight*, *KungFu3*, *KungFu4*, *Geinimi*, *GoldDream*, *KMin* and *Pjapps*.

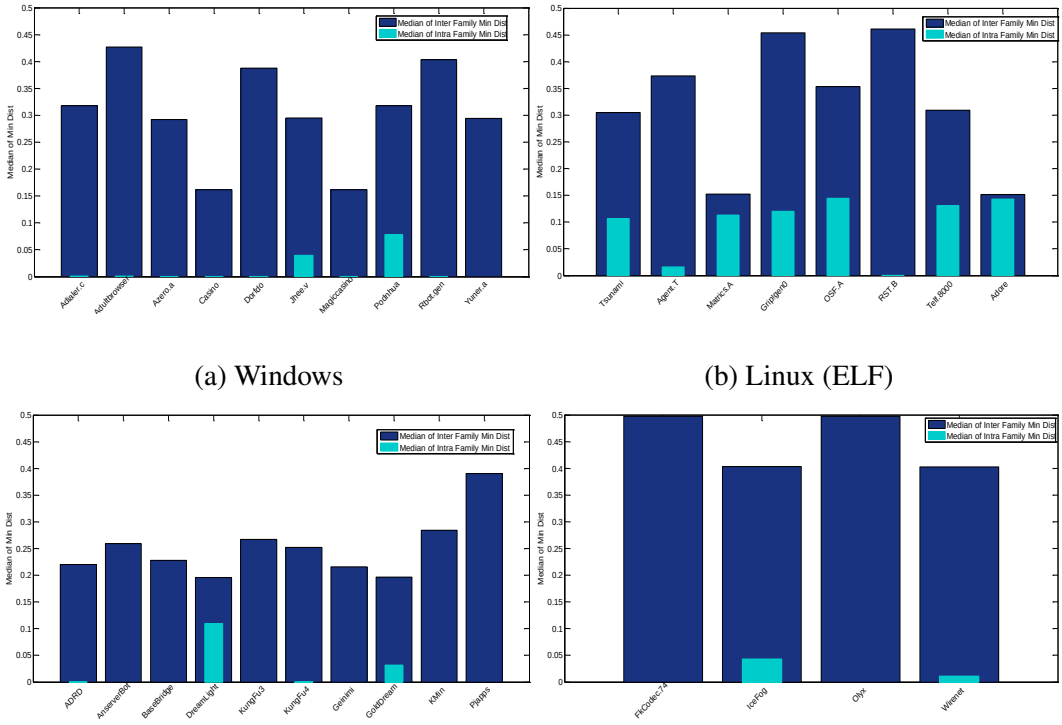
OS X Malware Dataset: Due to the shortage of OS X malware binaries in public, we used 4 different OS X malware families with each having 2 variants: *FkCodec.74*, *IceFog*, *Olyx* and *Wirenet*. These were obtained and pruned from [2].

Table 3.1: Maximum and minimum value of $\mathbf{m}_{\mathbf{intra}}^{\mathbf{min}}$ and $\mathbf{m}_{\mathbf{inter}}^{\mathbf{min}}$ across different families for Various OS and Packers

OS/Packer	Min $\mathbf{m}_{\mathbf{intra}}^{\mathbf{min}}$	Max $\mathbf{m}_{\mathbf{intra}}^{\mathbf{min}}$	Min $\mathbf{m}_{\mathbf{inter}}^{\mathbf{min}}$	Max $\mathbf{m}_{\mathbf{inter}}^{\mathbf{min}}$
Win/None	0	**0.0779**	0.1615	0.4268
Linux/None	0.0003	**0.1442**	0.1512	0.4608
Android/None	0	**0.1093**	0.1955	0.3906
OS X/None	0	**0.0433**	0.4029	0.4972
Win/UPX	0	**0.1619**	0.2173	0.4663
Win/FSG	0	**0.1921**	0.2063	0.4991
Win/NsPack	0.0001	**0.1802**	0.1842	0.5228
Win/PECompact	0	**0.1757**	0.1885	0.5047
Win/Polyene	0.0628	**0.1778**	0.2167	0.5281
Win/Telock	0.0353	0.1936	0.1899	0.7052
Win/Themida	0.0920	0.1747	0.1161	0.5399
Win/Upack	0	0.1991	0.1781	0.5367

Results: For every family, we compute the median of the Intra-Family Minimum Distances (Eqn. (3.14)) and Inter-Family Family Distances (Eqn. (3.15)). We repeat this for all 4 datasets from Various OS and the results are shown in Fig. 3.8. For all families in the 4 datasets, $\mathbf{m}_{\mathbf{intra}}^{\mathbf{min}}$ is larger than $\mathbf{m}_{\mathbf{inter}}^{\mathbf{min}}$. This shows that similarity is maintained within a family and there is also enough variation across families. In Tab. 3.1, we compute the maximum and minimum values of the medians. The maximum of $\mathbf{m}_{\mathbf{intra}}^{\mathbf{min}}$ for all families in all 4 datasets is always lesser than the minimum median value of $\mathbf{m}_{\mathbf{inter}}^{\mathbf{min}}$. This allows simple classifiers like thresholding to separate the families. The difference

is highest for OS X malware (**0.3596**), followed by Android (**0.0862**) and Windows (**0.0836**), and lowest for Linux (**0.007**).



(a) Windows

(b) Linux (ELF)

(c) Android

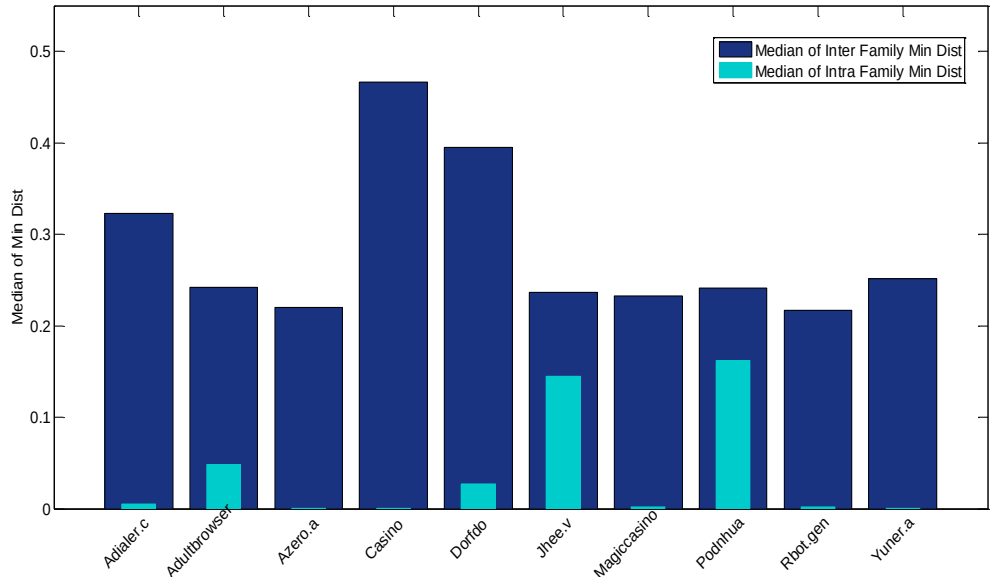
(d) OS X

Figure 3.8: Analysis on Unpacked Malware: We compare the median values of Intra-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$) and Inter-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$) for unpacked malware variants from various platforms: (a) Windows, (b) Linux, (c) Android and (d) OS X. For all case we see that $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$ is lesser than $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$, thus showing that there is similarity within families and not across families.

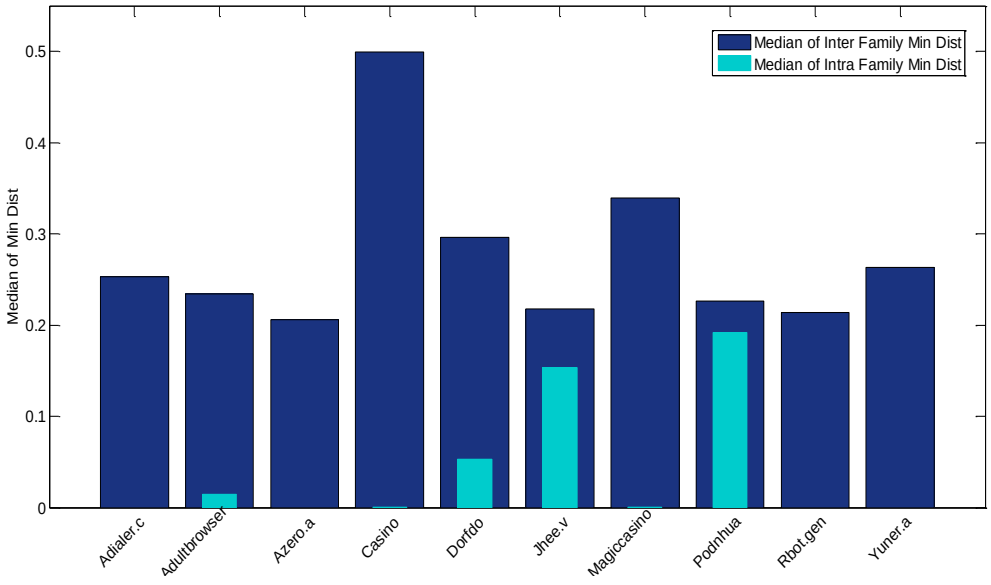
3.4.5 Analysis of Packed Malware

Most of the malware observed today are packed. To evaluate our methodology on packed malware, we pack the Windows malware dataset using 8 packers: *UPX*, *FSG*, *NsPack*, *PECompact*, *Polyene*, *Telock*, *Themida* and *Upack*. These 8 packers can be grouped into 4 categories [52]: Compressors (*UPX*, *FSG*, *NsPack*, *PECompact*, *Upack*), Cryptors (*Polyene*), Multi-layer cryptors (*Telock*) and Virtualization-based packers (*Themida*). After packing, the packed families for every packer are treated as new families, example *Adialer.C.UPX, Adultbrowser.UPX*. This allows us to measure the similarity among the packed variants within the families of a specific packer. In Sec. 3.5, we will see how packed variants can be differentiated both across families and packers.

Results: Similar to the analysis of unpacked malware, we compute $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$ and $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$ of of the packed variants for every family per packer. Fig. 3.9 shows the results for *UPX* and *FSG*, Fig. 3.10 for *NsPack* and *PECompact*, Fig. 3.11 for *Polyene* and *Telock*, and Fig. 3.12 for *Themida* and *Upack*. For all packers, $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$ for a family is always lesser than $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$ for that family. In Tab. 3.1, we see that the maximum value of $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$ is greater than the minimum value of $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}(i)$ for *UPX*, *FSG*, *NsPack*, *PECompact* and *Polyene*. In the case of *Telock*, *Themida* and *Upack*, there are some families whose maximum value of $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$ is lesser than the minimum value of $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$ and a simple threshold based classifier will not work for these classifiers. However, since the individual median values of $\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$ of all families are lesser than the median of $\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$, we will use Nearest Neighbor based classifier to classify families (Sec. 3.5).

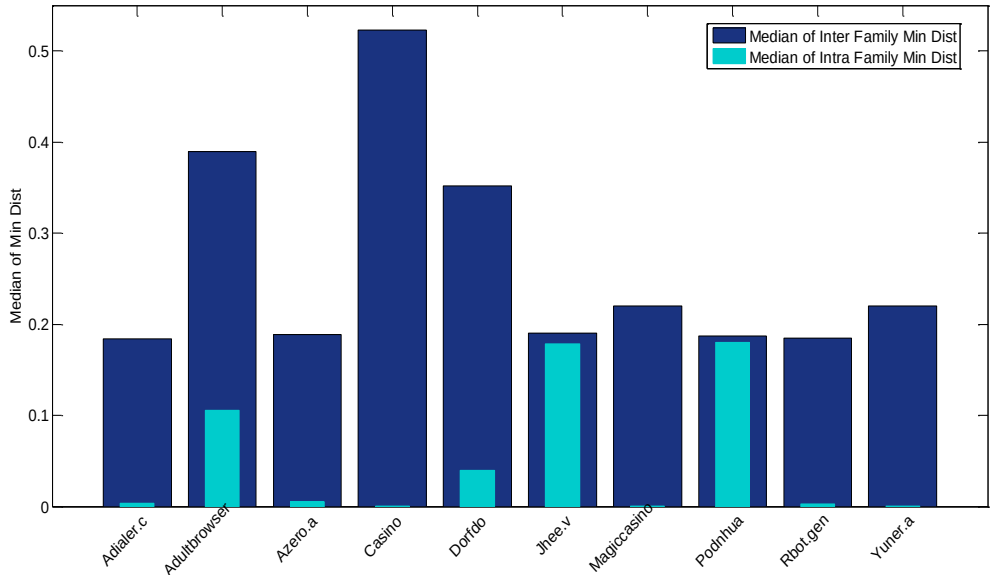


(a) UPX

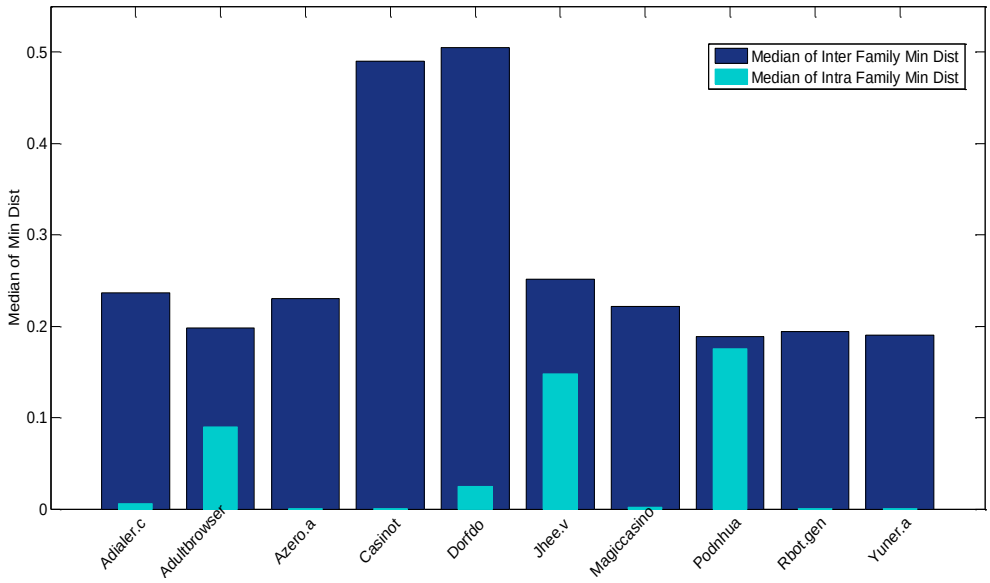


(b) FSG

Figure 3.9: Analysis on Packed Malware: We compare the median values of Intra-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$) and Inter-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$) for packed malware variants from various packers: (a) UPX, (b) FSG

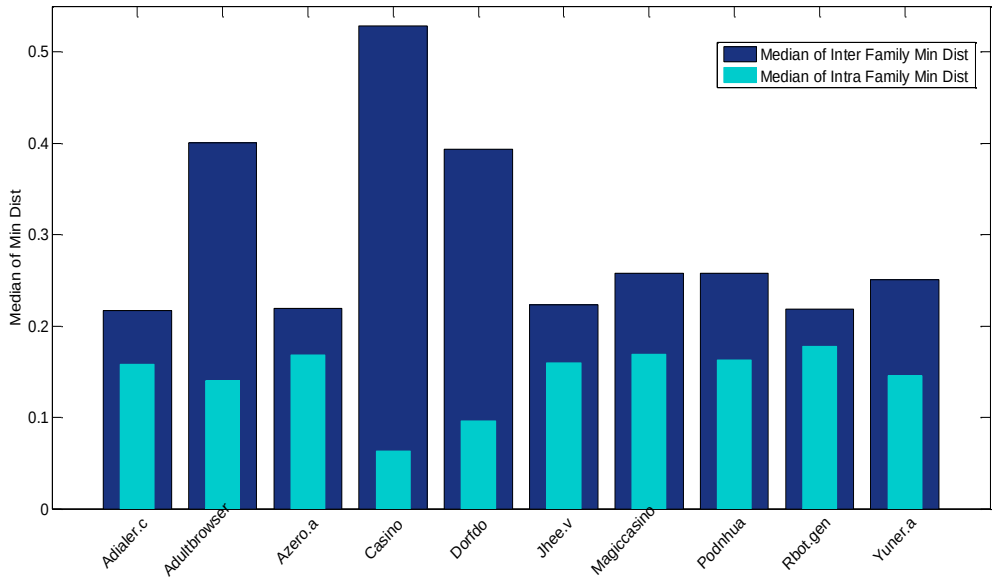


(a) NsPack

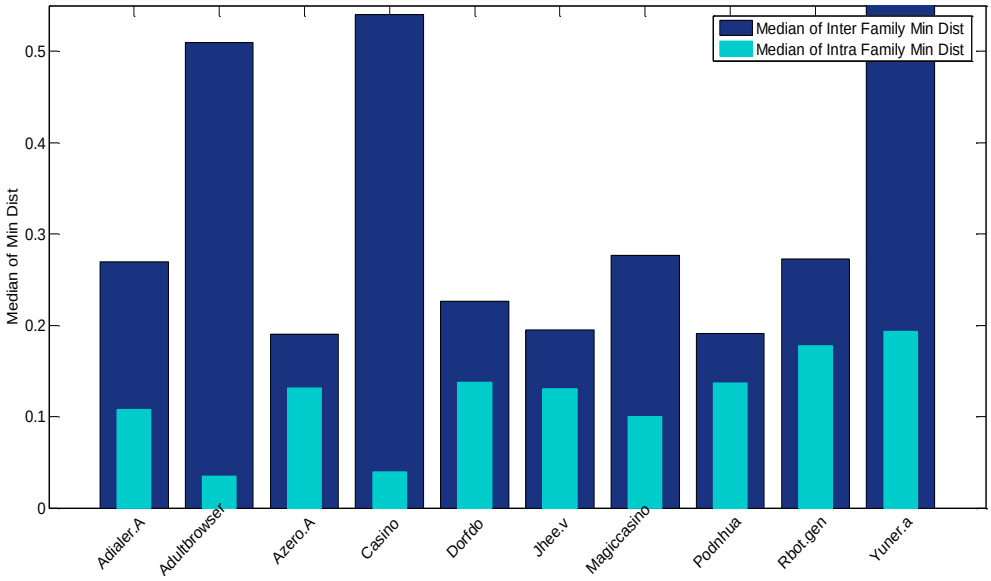


(b) PECompact

Figure 3.10: Analysis on Packed Malware: We compare the median values of Intra-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$) and Inter-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$) for packed malware variants from various packers: (a) NsPack, (b) PECompact

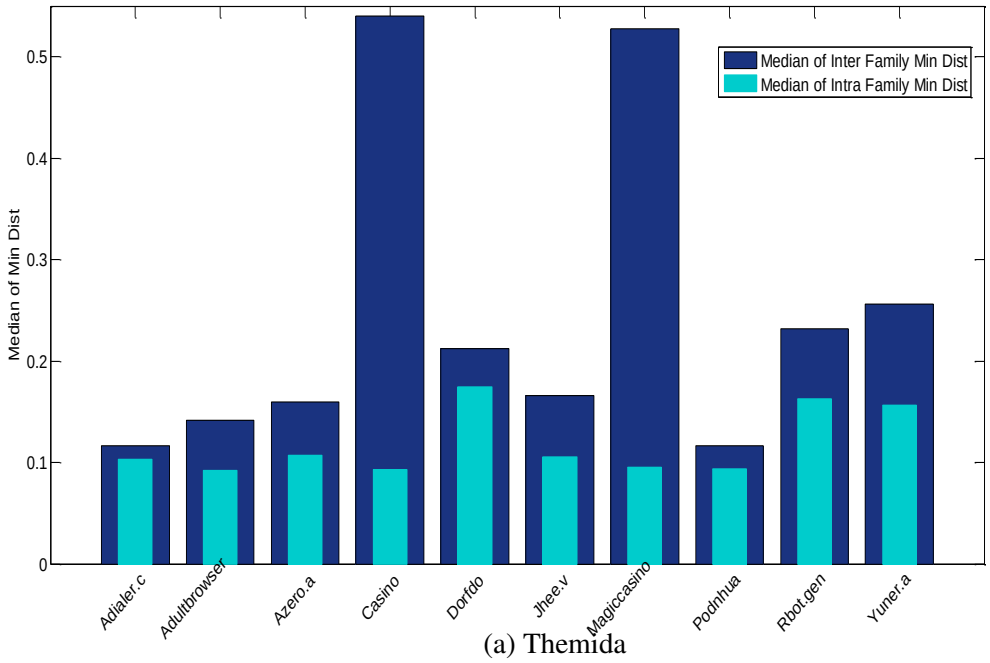


(a) Polyene

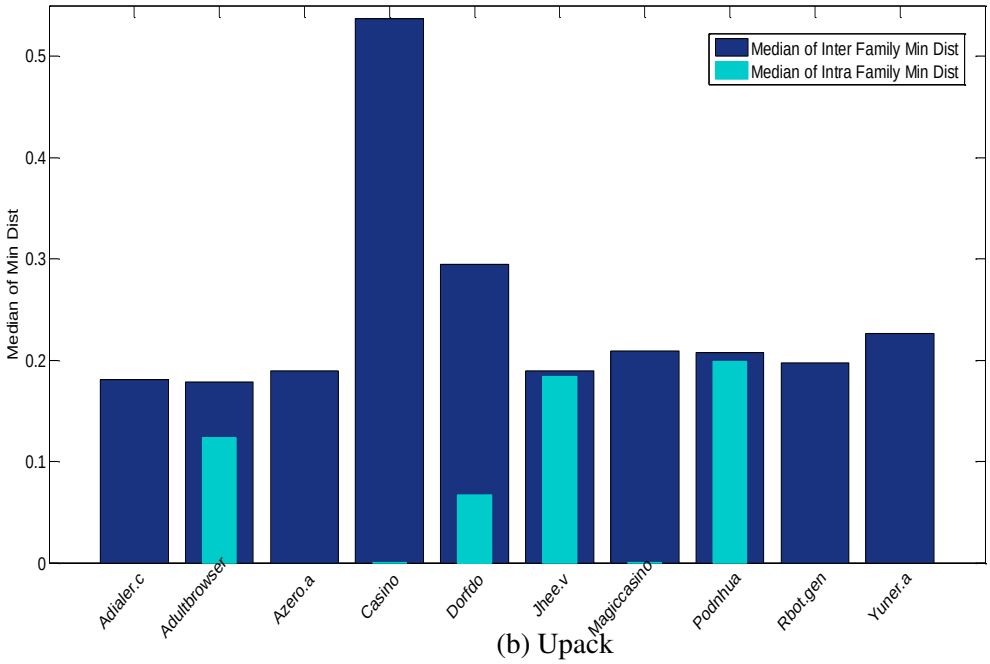


(b) Telock

Figure 3.11: Analysis on Packed Malware: We compare the median values of Intra-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$) and Inter-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$) for packed malware variants from various packers: (a) Polyene, (b) Telock



(a) Themida



(b) Upack

Figure 3.12: Analysis on Packed Malware: We compare the median values of Intra-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{intra}}$) and Inter-Family Minimum Distances ($\mathbf{m}^{\mathbf{min}}_{\mathbf{inter}}$) for packed malware variants from various packers: (a) Themida, (b) Upack

3.5 Malware Classification

Here we conduct several experiments and show how malware variants can be classified according to their families. We start with small datasets of unpacked malware from various platforms and then evaluate on packed malware datasets. Then we test on publically available malware datasets belonging to platforms of Windows, Android and Linux. Finally, we evaluate our method on a large malware corpus.

For all our experiments, we perform supervised classification with 10-fold cross validation and compute the average classification c_{acc}. We use Nearest Neighbor (NN) classifier which assigns the family of the nearest malware to an unknown malware. For *Generalized Features*, we use GIST with the following parameters: $S_q = \mu, (b_x, b_y) = (4, 4), N_f = 20$ to obtain a 320 dimensional feature vector. All images are resized to 64×64 before computing the features. The effect of various parameters and other similarity features are also evaluated.

3.5.1 Classification of Small Scale Datasets

Unpacked Malware: We first evaluate our method on the unpacked malware datasets from different platforms for which we performed the similarity analysis. We obtained a c_{acc} of 100% for **Windows Malware Dataset**, 93.75% **Linux Malware Dataset**, and 99% for the **Android Malware Dataset**. This shows the there are samples in some of **Linux Malware Dataset** which are closer to other families. We did not evaluate on **OS X Malware Dataset** since the number of samples was too low.

Packed Malware: Next we evaluate on the packed malware datasets that were packed using 8 packers: *UPX*, *FSG*, *NsPack*, *PECompact*, *Polyene*, *Telock*, *Themida* and *Up-*

ack. A popular misconception is that if two binaries belonging to different families are packed using the same packer, then the two binaries are going have similar characteristics. While this is true for static code analysis, this does not hold for their structural characteristics as we see in Fig. 3.4. Images of malware variants belonging to different families but packed with the same packer are indeed different. Hence, we treat the variants of a family coming from a particular packer as a new family (for example: *Adialer.C.UPX*, *Adialer.C.PeC*, *Adialer.C.Upack*). We repeated the classification for experiments for the 8 different packers and obtained a c_{acc} of 100% for all packers except *Upack*, for which c_{acc} was 99%.

Mixing Unpacked and Packed Malware: Now, we include unpacked malware families and the corresponding packed families in a single dataset repeat the experiments. For 10 families with each having 10 variants and packed with 8 different packers, we obtain a total of 900 samples divided among 90 families (10 unpacked and 80 packed families). We obtained a c_{acc} of 99.44% which shows that similarity is preserved in datasets which contain both unpacked and packed malware variants. From the confusion matrix (Fig. 3.13), we see that the packed variants neither get misclassified with packed variants from other packers nor with the unpacked variants from which they were generated from. Almost all families have an accuracy of 100% except a few which are packed using *Upack* packer. Based on these results from our controlled experiments, we will evaluate our technique on real world malware datasets in Sec.3.5.2.

3.5.2 Classification of Public Datasets

Malimg Dataset: The Malimg dataset contains 25 malware families with 9,342 samples, which we obtained from the authors of [72]. The dataset has a mixture of both

packed and unpacked malware and the number of samples per family varies from 80 to 2,949. The families were labeled using a single Antivirus software. We obtained a c_{acc} of 97.4% on this dataset.

Malheur Dataset: The Malheur dataset consists of 3,131 malware binaries from 24 malware families, which we obtained from the authors of [86]. The malware binaries were labeled such that a majority amongst six different Antivirus products shared similar labels. The number of samples per family varied between 20 and 300. For this dataset, we obtained a c_{acc} of 98.37%.

Android MalGenome Dataset: This dataset is part of the Android Malware Genome Project [114] comprising over 1,200 Android malware among 49 families. We chose families that had atleast 20 samples for our analysis. This yielded 1094 samples divided among 13 families and we obtained a c_{acc} of 84.55%.

VxShare ELF Malware Dataset: We obtained this dataset from VxShare [18]. It comprised over 2,000 Linux malware which we labeled using an Antivirus software. We chose families that had atleast 20 samples which yielded 568 samples divided among 8 families. We obtained a c_{acc} of 83.27% on this dataset.

3.5.3 Comparison with Other Image based Similarity Features

We compare the GIST features based on texture and spatial layout with other similarity features, namely, the Homogenous Texture Descriptor (HTD) and the Color Layout Descriptor (CLD). The HTD is a 96-dimensional texture based image similarity descriptor where an image is filtered over 48 sub-bands after which the mean and standard deviation on each filtered image are grouped to form the feature vector. The CLD,on the other hand, is a layout based descriptor. The image is divided into an 8x8 grid and

the mean value of every grid block is computed to obtain an 8x8 matrix. The Discrete Cosine Transform (2D) is computed on this matrix and the first n (17 in our case) coefficients form the descriptor. Apart from these, we also consider GIST features on different number of sub-bands, namely 20 and 3 (GIST-320 and GIST-48).

Varying Training dataset size: We vary the size of the training set keeping the number of nearest neighbors constant and repeat the experiments. As shown in Fig. 3.14, the GIST features based on texture and layout outperforms both the texture based HTD and the layout based CLD image similarity descriptors. As expected, the classification accuracy decreases as the training sample size decreases.

Varying number of nearest neighbors: Here, we fix the training dataset size to 50% and vary the nearest neighbors. Once again, GIST outperforms all other features (Fig. 3.15). The accuracy is maximum for k=1 and decreases as the number of nearest neighbors increases.

Classification Time: Next, we report the time needed to classify a sample as we vary the training dataset size (Fig. 3.16). We fix the number of nearest neighbors as 1. As expected, the features with low dimensions (CLD, GIST-48) require less time and the GIST-320 requires the most time to be classified. Also, this time linearly increases with increase in training dataset size.

3.5.4 Comparison with Static Analysis

We compare our technique with a recently proposed static analysis based on the distribution of bigrams. The code section of a malware is dumped and the distribution of its bigrams is computed over a bit shifted window to obtain a 65,536 dimensional feature vector. We repeated the experiments using this feature vector and the results are

Table 3.2: Average computation time

	Static Feature	Image Feature
Feature Length	65536	384
Classification Accuracy	0.9124	0.9830
Computation Time	6 s	60 ms

shown in Tab. 3.2. We see that the classification accuracy is almost the same. However, the image based feature is around 100 times faster than bigrams based static analysis. Also, in terms of storage, the image feature requires 384 features while the bigrams based feature requires 65536, close to 170 times more.

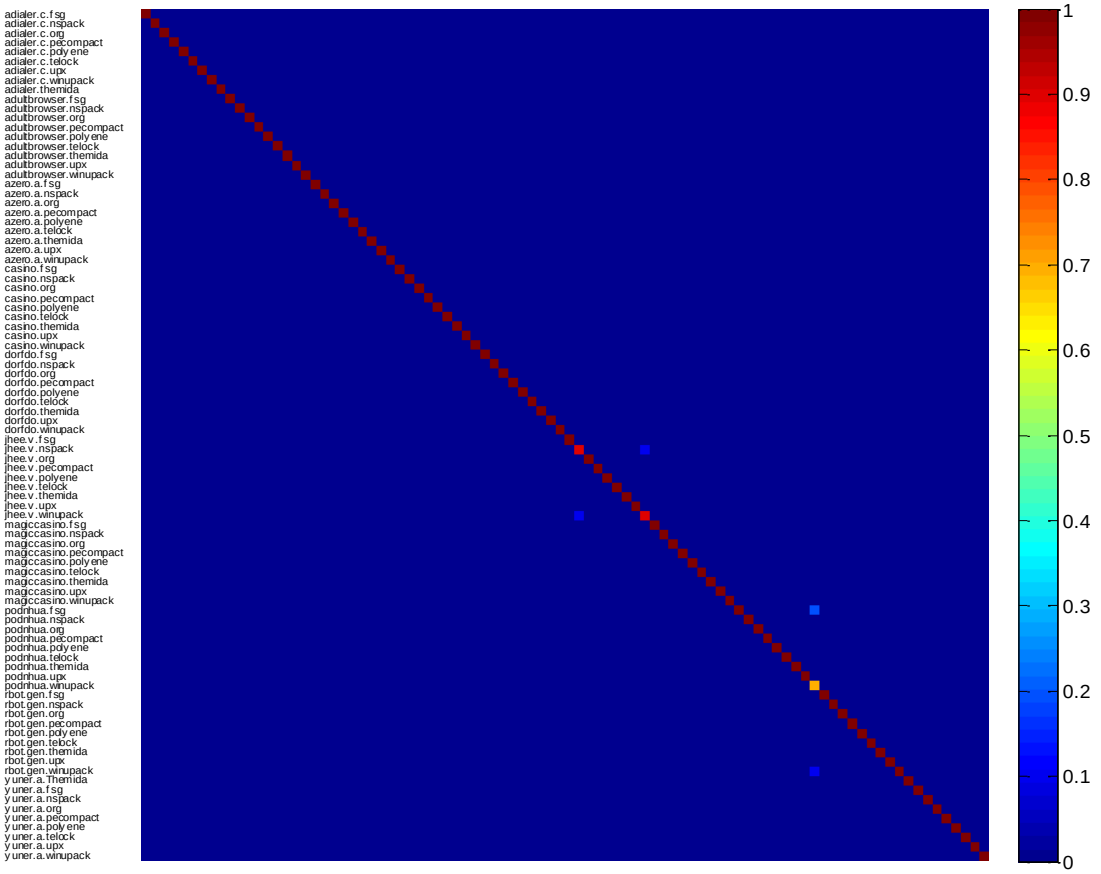


Figure 3.13: Confusion matrix on both Unpacked and Packed malware families. Unpacked variants from 10 malware families (*Adialer.c*, *Adultbrowser*, *Azero.a*, *Casino*, *Dorfdo*, *Jhee.v*, *Magiccasino*, *Podnhua*, *Rbot.gen* and *Yuner.a*) are packed using 8 packers: *UPX*, *FSG*, *NsPack*, *PECompact*, *Polyene*, *Telock*, *Themida* and *Upack*. Each family has 10 variants thus resulting in 900 malware (100 unpacked and 800 packed). Unpacked are suffixed with '.org' (example *Adialer.c.org*). On doing supervised classification with 10-fold cross validation, the average classification accuracy (c_{acc}) obtained was 99.44%. This shows that after packing, packed variants can be treated as new families and they will not get misclassified with packed variants from other packers or with the original unpacked variants from they were generated from.

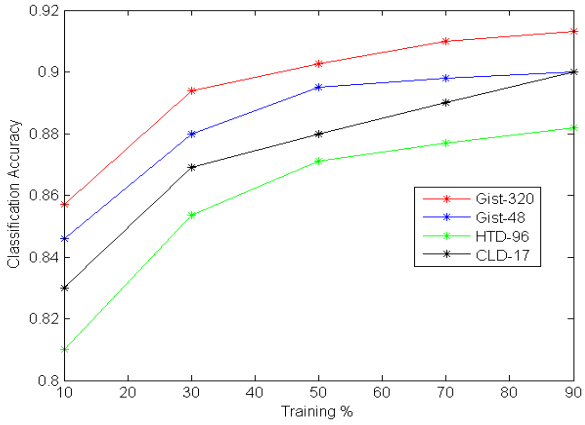


Figure 3.14: Varying Training Data Size

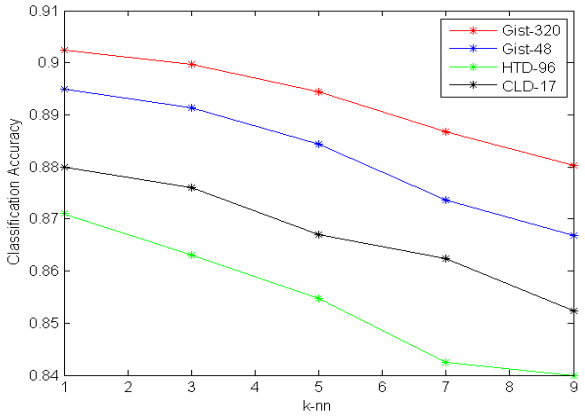


Figure 3.15: Varying k-Nearest Neighbors

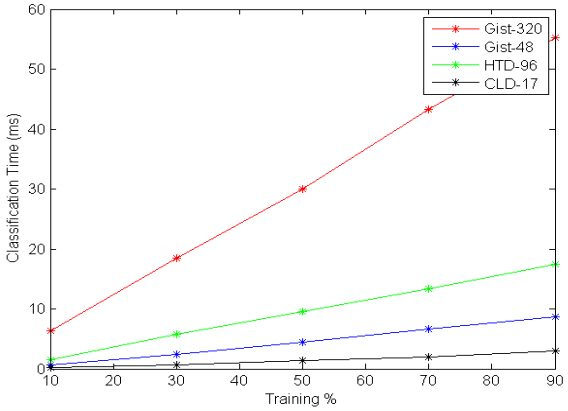


Figure 3.16: Classification Time

3.5.5 Comparison with Dynamic Analysis

In dynamic analysis, the behavior of the malware is traced by running the malware executable in a sandbox environment for several minutes. Based on the sandbox system under consideration, the type and granularity of data that is recorded might be different.

Here, we evaluate two different dynamic analysis approaches. The first and commonly used approach is system-call level monitoring which might be implemented using API hooking or VM introspection. The system generates a sequential report of the monitored behavior for each binary, based on the performed operations and actions. The report typically includes all system calls and their arguments stored in a representation specifically tailored to behavior-based analysis.

The second approach, called forensic snapshot comparison relies on a combination of features that are based on comparing the pre-infection and post-infection system snapshots. Some of the key features collected include AUTORUN ENTRIES, CONNECTION PORTS, DNS RECORDS, DROP FILES, PROCESS CHANGES, MUTEXES, THREAD COUNTS, and REGISTRY MODS. A whitelisting process is used to weigh down certain commonly occurring attribute values for filenames and DNS entries. A key difference between system-call monitoring and forensic comparison, is that the latter approach does not capture the temporal ordering of forensic events. The advantage is that it is simpler to implement and prioritize events that are deemed to be most interesting. To identify deterministic and non-deterministic features, each malware is executed three different times on different virtual machines and a JSON object is generated describing the malware behavior in each execution [113]. Depending on the presence of the key features, a binary feature vector is generated from the JSON object. If a key feature is present in the JSON object, then corresponding component

in the feature vector is 1; otherwise the component is 0. The size of the feature vector depends on the diversity of the behavior of the malware corpus, but varies between 600-2000 for most datasets we have used.

Evaluation: In this section we present several experiments that compare the results of the image similarity analysis strategy against datasets analyzed in three separate malware binary classification efforts. In each experiment, every malware binary is characterized by a feature vector. We then use k-nearest neighbors (k-NN) with Euclidean distance measure for classification. The feature vectors are computed for all binaries in the datasets and are divided into a training set and a testing set. For every feature vector in the testing set, the k-nearest neighbors in the training set are obtained. Then the vector is classified to the class which is the mode of its k-nearest neighbors. For all tests, we do a 10-fold cross validation, where under each test, a random subset of a class is used for training and testing. For each iteration, this test randomly selects 90% data from a class for training and 10% for testing.

We summarize our datasets in Table 3.3. In the first experiment, Section 3.5.6, we compare image similarity analysis against a 2140 sample dataset. Out of this 2140, we downselected 393 binaries that have consistent labels and sufficient membership. We call this subset the *Host-Rx* reference dataset which was classified using a dynamic analysis technique where the binaries are clustered based on their observed forensic impacts on the host [113]. These clusters are also examined against antivirus labels, and the minimum family sample size is 20. We employed both image analysis and dynamic analysis on the Host-Rx dataset, recomputing the dynamic features and comparing the classification performance, since clustering is mainly focused in [113]. The average classification accuracy is 98% and 95% for dynamic and image analysis, respectively.

Table 3.3: Malware datasets

Dataset	Num Binaries	Num Families
Host-Rx Reference Dataset	393	6
Host-Rx Application Dataset	2,140	-
Malhuer Reference Dataset	3,131	24
Malheur Application Dataset	33,698	-
VXHeavens Dataset	63,002	531
Anubis Dataset	685,000	1441
Offensive Computing Dataset	1.4 M	2140

Thus, we find the image similarity technique achieves comparable success but is able to complete its feature analysis in approximately 1/4000 the time required per binary.

In the second experiment, we examine the *Malhuer* dataset, which was originally classified based on features derived by dynamic system call interception, as presented in [86]. These execution traces were converted to a Malware Instruction Set (MIST) format, from which feature vectors were extracted to effectively characterize the binaries. Since [86] already reports the results of a malware classification based on these features, we do not repeat the dynamic analysis. In Section 3.5.7, we present a comparative assessment of the Malheur dataset, which includes both a reference data set of 3131 malware binaries comprising 24 unique malware families, and an application dataset of roughly 33 thousand binaries that range from malicious, to unknown, to known benign. Using these datasets, we found that image similarity classification produced comparable accuracies in both datasets, ranging from 97% for the reference set, to 86% accuracy for the application set.

Finally, we present large scale experiments using three datasets: *VX Heavens* dataset, Anubis Dataset and Offensive Computing Dataset.

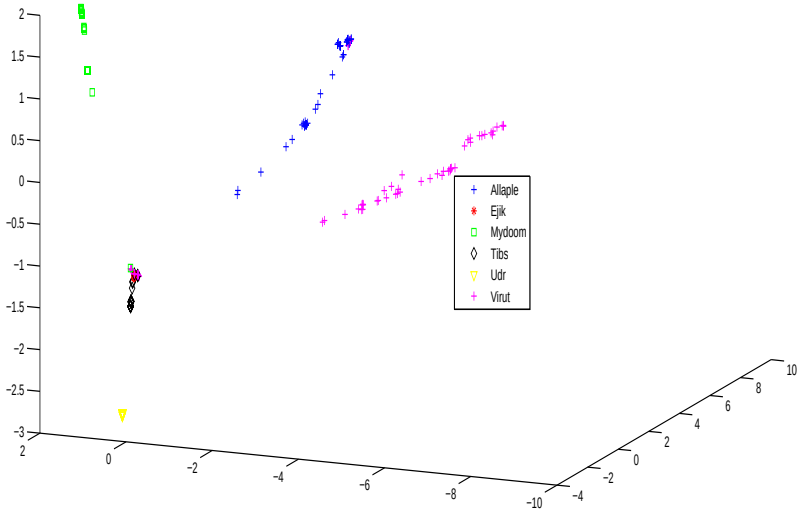


Figure 3.17: Lower dimensional visualization of dynamic features on Host-Rx dataset

3.5.6 Experiment 1: Image Similarity Analysis vs Host-Rx Dynamic Analysis Dataset

In this experiment, we compute the feature vectors for image similarity analysis and forensic dump-based dynamic analysis. We use the Host-RX dataset [113], for this comparison.

The following process was used to select malware instances with reliable labels. The initial malware corpus consisted of 2,140 malware binaries. The AV labels for these binaries were obtained from Virustotal.com [19]. Six AV vendors were used: AVG, AntiVir, BitDefender, Ikarus, Kaspersky and Microsoft. We attribute a label to a malware instance if at least 2 AV vendors share similar labels. Out of 2140 instances, only 450 malware samples had consistent labels. However, some malware families had very few samples. To conduct a meaningful (representative) analysis of the features collected, each family should have a large enough group of exemplars from which to

Table 3.4: Family/packer summary: Host-Rx reference dataset

Family	Total	Packed	Packer
Allaple	81	3	UPX
Ejik	30	11	10 PeC, 1 nsPack
Mydoom	151	124	UPX
Tibs	31	0	-
Udr	34	34	AsPack
Virut	66	3	UPX

derive consistent features. For consistency with previous classification research ([86]) that is compared in the next section, we chose to remove families with less than 20 samples, and compiled a final collection of 393 malware binaries. These binaries (shown in Tab. 3.4) represented 6 malware families. We also check if these malware are packed or not using PeID and report the labels in Tab. 3.4. However, some of the malware that PeID did not detect could also be packed.

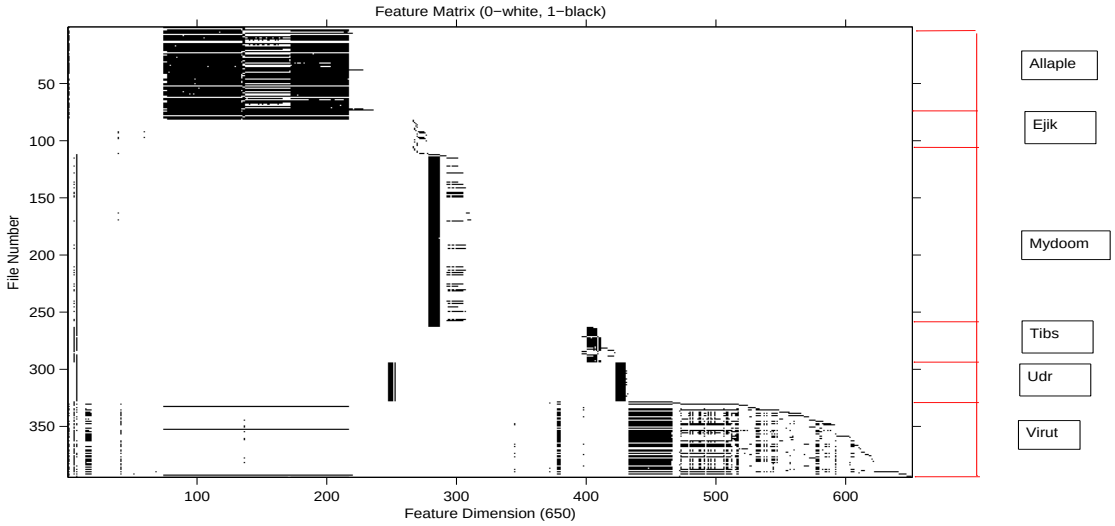


Figure 3.18: Feature matrix of dynamic analysis features on Host-Rx dataset

Dynamic Analysis Features: The dynamic analysis features are obtained from the

forensic memory dumps and then converted to a vectorial form to produce a 651 dimensional vector for every malware instance, as shown in Fig. 3.18. The result is a binary matrix, where 1 refers to the presence of a corresponding feature. The first 81 samples belong to Allaple and most of their features are between 75 and 219 (Fig. 3.18). Similarly, we observe that other families like Ejik, Mydoom, Tibs, Udr and Virut have features concentrated in different sections of the feature matrix. For better visualization, these 651 dimensional features are projected to a lower dimensional space using multidimensional scaling [10]. As shown in Fig. 3.17, the 6 families cluster well.

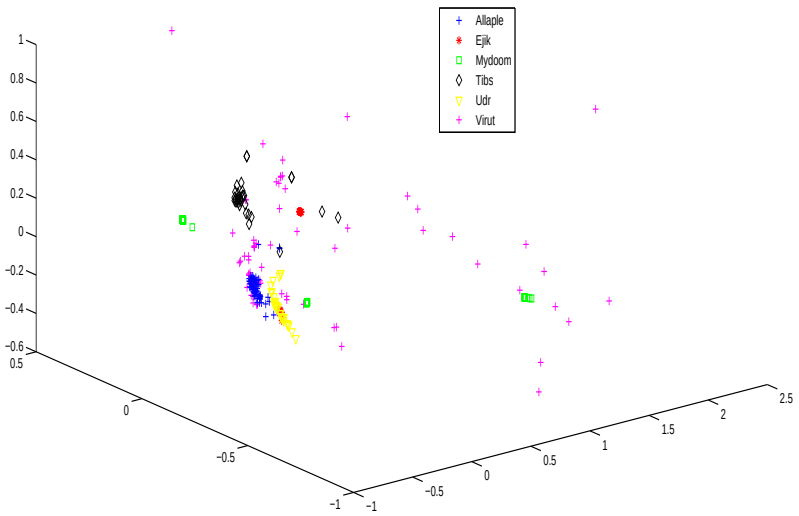


Figure 3.19: Lower dimensional visualization of image similarity features on Host-Rx dataset

To further validate the dynamic features, we classify the malware using a k-NN based classification across a 10-fold cross validation. For k=3, we obtain an average classification accuracy of 98.22%. As shown in Tab. 3.5, 387 out of 394 samples get classified correctly. Four samples of Virut are misclassified as Allaple due to feature similarity with that of Allaple (Fig. 3.18).

Table 3.5: Confusion matrix for dynamic features on Host-Rx dataset

	Allaple	Ejik	Mydoom	Tibs	Udr	Virut
Allaple	81	0	0	0	0	4
Ejik	0	30	1	1	0	1
Mydoom	0	0	150	0	0	0
Tibs	0	0	0	30	0	0
Udr	0	0	0	0	34	0
Virut	0	0	0	0	0	61

Image Similarity Features: Every malware binary is converted to an image and the GIST feature is computed. Hence every malware is characterized as a 384 dimensional vector which is based on the texture and layout of the malware image. Fig. 3.19 illustrates a lower dimensional visualization of the static features. Some malware families, such as Ejik, Allaple, Udr and Tibs, are tightly clustered. Mydoom exhibits an interesting pattern, where three sub-clusters emerge, and each is tightly formed. We manually analyzed each sub-cluster of Mydoom and found that the first sub-cluster had 124 samples, all of which were packed using UPX. The second and third sub-clusters had 15 and 12 members, respectively, but the images inside these sub-clusters appeared similar among themselves but different from the images of other sub-clusters. In contrast to the other families, the Virut family was not tightly clustered, as most images of the Virut family were found to be dissimilar.

Similar to dynamic analysis, we performed a k-NN based classification across a 10-fold cross validation using the image based features. For k=3, we obtained an average classification accuracy of 95.14%. The confusion matrix is shown in Table 3.6. Except for Allaple and Virut families, almost all samples of the other four families were accurately classified.

Table 3.6: Confusion matrix for image similarity features on Host-Rx dataset

	Allaple	Ejik	Mydoom	Tibs	Udr	Virut
Allaple	74	0	0	0	0	4
Ejik	0	30	0	0	0	2
Mydoom	0	0	151	0	0	2
Tibs	0	0	0	30	0	4
Udr	0	0	0	0	34	0
Virut	7	0	0	1	0	54

Size of training set: We evaluate image similarity anslysis and dynamic analysis by varying the number of training samples. We fix k = 3. The test is repeated three times and the average classification accuracy is obtained for each test, and then average again. Fig. 3.20 illustrates that the dynamic features outperform the image similarity features when the number of training samples are fewer (10%, 30%). Using 50% - 90% training samples, the difference is only marginal.

Varying k: Next, we fix the percentage of training samples at 50% per family and explore the impact of varying k. Similar to the previous test, the dynamic features are better at higher values of k (Fig. 3.21, but there is only a marginal difference at lower values of k. This is also evident from Fig. 3.17, where we see that the dynamic features are more tightly clustered when compared to the image similarity features (Fig. 3.19).

Computation Time: The primary advantage of image similarity features is the computation time. Since, our image similarity features are a direct abstraction of the raw binaries and do not require disassembly, the time to compute these features are considerably lower. We used an unoptimized Matlab implementation to compute the GIST features ([5]) on an Intel(R) Core(TM) i7 CPU running Windows 7 operating system. The average time to convert a malware binary to an image and compute the GIST fea-

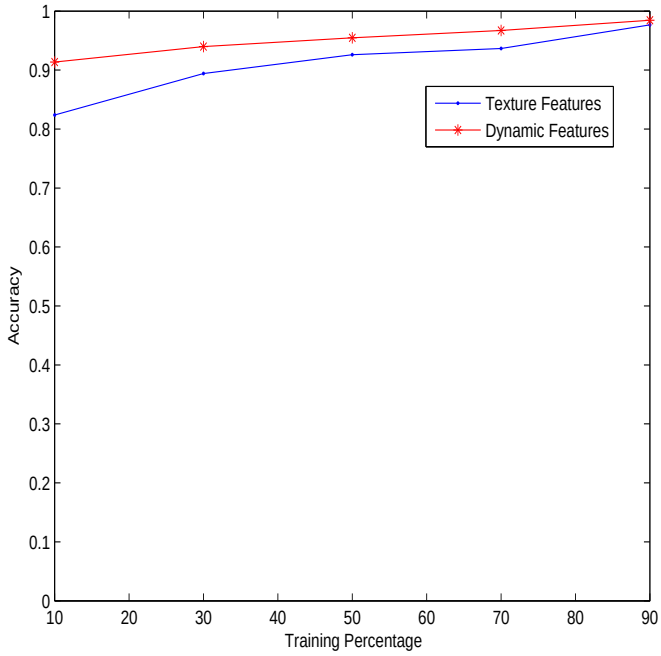


Figure 3.20: Comparing classification accuracies of image similarity features and dynamic analysis features by varying number of training samples on Host-Rx Dataset

Table 3.7: Average computation time

Dynamic Feature	Image Similarity Feature
4 mins	60 ms

tures was **60 ms**. In contrast, the average time required to obtain dynamic features from the same binary was **4 minutes**. As shown in Table 3.7, the image similarity features are about **4000 times** faster.

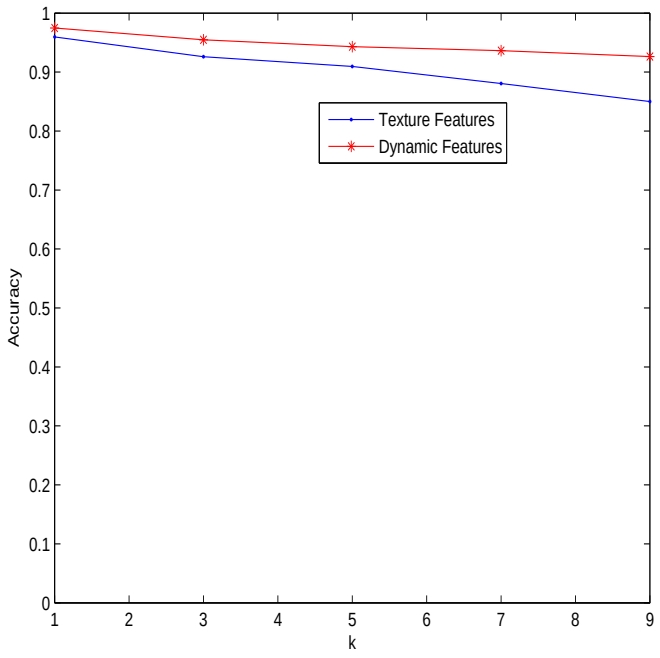


Figure 3.21: Comparing classification accuracies of image similarity features and dynamic analysis features by varying k on Host-Rx Dataset

3.5.7 Experiment 2: Image Similarity Analysis vs Malheur Dynamic Analysis Dataset

For further validation, we compare image similarity analysis to another dynamic analysis technique that derives its features from system call intercepts [86]. Here, we employ the Malheur dataset, which includes both a reference dataset and an application dataset. We obtained both dataset and classification results from the researchers who performed a dynamic analysis using this corpus [86]. We do not repeat their dynamic analysis here, but rather focus on performing a image similarity analysis on this dataset, and then compare results from both methods.

The Malheur [86] reference dataset consists of 3131 malware binaries from instances of 24 malware families. The malware binaries were labeled such that a major-

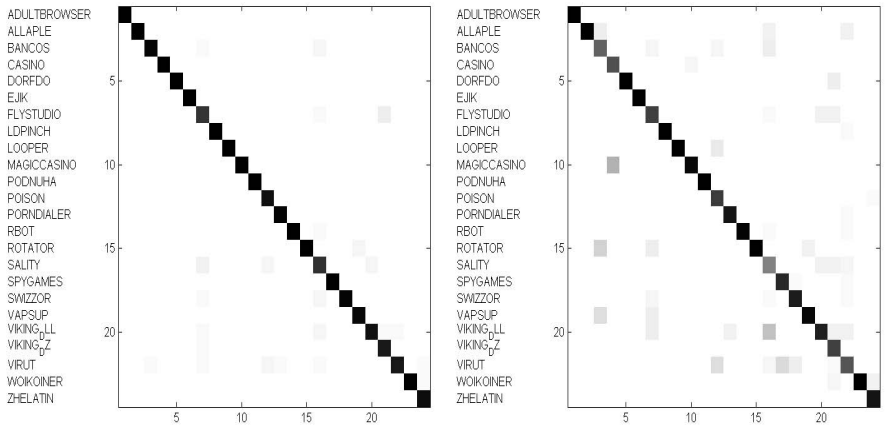


Figure 3.22: Confusion matrix obtained on Malheur Reference dataset using: (a) 10-fold cross validation, (b) 10% training sample

ity amongst six different antivirus products shared similar labels. Further, in order to compensate for skewed distribution of samples per family, the authors of [86] discarded families less than 20 samples, and restricted the maximum samples per family to 300. The exact number of samples per family and the number of malware packed are given in Tab. 3.8. Once again we identify the packers using PeID. Although it is known that PeID could sometimes miss some packers, we still go with these labels.

For the image similarity analysis, we repeated the experiments by converting these malware to images, computing the image features and then classifying them using k-NN based classification. Initially, we do a 10-fold cross validation: the confusion matrix is shown in Fig. 3.22. We obtained an average classification accuracy of 97.57%. In [86], the authors obtained similar results using dynamic analysis. From Tab. 3.8, we see that many families are packed, and some families such as Adultbrowser, Casino, Magiccasino and Podnhua, are all packed using the same packer, viz. UPX.

A popular misconception is that if two binaries belonging to different families are

packed using the same packer, then the two binaries are going to appear similar. In Fig. 3.23, we can see that the images of malware binaries belonging to different families but packed with the same packer are indeed different.

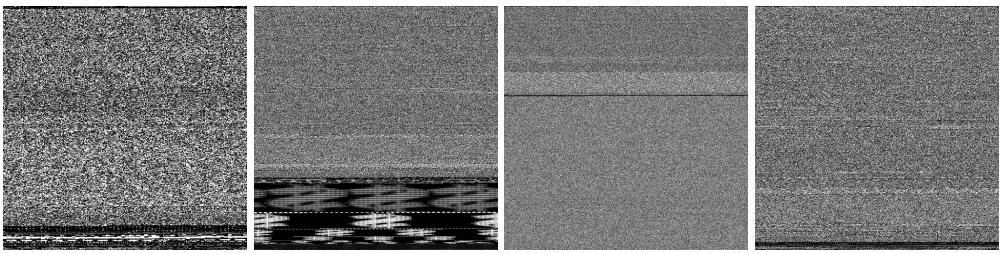


Figure 3.23: Images of malware binaries packed with UPX belonging to (a) Adult-browser, (b) Casino, (c) Flystudio, (d) Magiccasino

Instead of doing 10-fold cross validation, we randomly chose 10% of the samples from each family for training and the rest for testing. This was repeated multiple times. The average classification accuracy only dropped to 91.30% and the confusion matrix is shown in Fig. 3.22.

Malheur Application Dataset: This dataset consists of unknown binaries obtained over a period of seven consecutive days from AV vendor Sunbelt Software. We received a total of 33,698 binaries. However, the authors of [86] labeled these using Kaspersky Antivirus. Out of 33,698 binaries, 7612 were labeled as 'unknown'. The authors mention that these are a set of benign executables.

In performing image similarity analysis on the Application Dataset, we retained the same labels given by the authors of [86]. Fig. 3.22 shows the confusion matrix we obtained. The row with maximum confusion corresponds to the set of 'Nothing Found' as mentioned in [86]. This set consisted of 7,622 binaries and were labeled as being "benign" in [86]. However, when we scanned these binaries using Microsoft

Security Essentials, 3,393 binaries were flagged as malicious (our experiments also confirm this). We then repeated the experiment by not considering this mixed set. We obtained an average classification accuracy of 86.15% (Fig. 3.22).

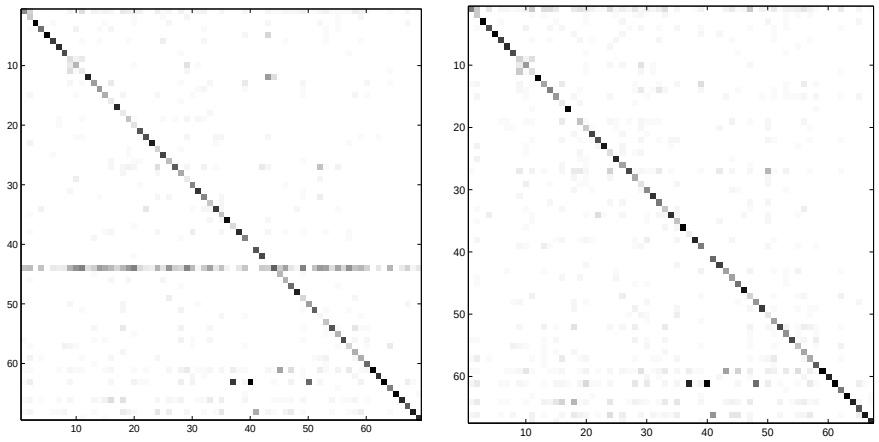


Figure 3.24: Confusion matrix on (a) Malheur Application data, (b) Malheur Application data without "benign" executables.

3.5.8 Large Scale Experiments

VX-Heavens Application Dataset: We performed image similarity analysis on a larger corpus, consisting of 63,002 malware from 531 families (as labeled by Microsoft Security Essentials). These malware were obtained from VX-Heavens [20]. The results are shown in Fig. 3.25. The average classification accuracy we obtained was 72.8%. 33 families were classified with an accuracy of 100%. Some of these include Skintrim.N,Yuner.A, Rootkit.AFE, Autorun.K, Adrotator. A total of 105 families fell above 90% accuracy, including Vake.H, Lolyda.AT, Seimon.D, Swizzor.gen!I, Azero.A, Allaple.A, Alueron.d, Instantaccess, Dialplatform, Jhee.V, Startpage.DE, to name a few. The families with high classification accuracies are those for which the

Table 3.8: Family/packer summary: Malheur Reference dataset

Family	Total	Packed	Packer
Adultbrowser	262	262	UPX
Allaple	300	0	-
Bancos	48	0	-
Casino	140	140	UPX
Dorfdo	65	0	-
Ejik	168	168	PECompact
Flystudio	32	2	UPX
Ldpinch	43	0	-
Looper	209	209	Aspack
Magiccasino	174	174	UPX
Podnhua	300	300	UPX
Poison	26	25	PEncrypt
Porndialer	97	0	-
Rbot	101	0	-
Rotator	300	0	-
Sality	84	0	-
Spygames	139	0	-
Swizzor	78	0	-
Vapsup	45	0	-
Viking_dll	158	0	-
Viking_dz	58	58	FSG
Virut	202	0	-
Woikoiner	50	0	-
Zhelatin	41	0	-

variants appear visually similar. However, there are also families with low classification accuracies. 23 families had classification less than 10%. These comprised a total of 1,061 binaries (about 1.6% of the dataset). Some example families include Orsam.RTS, Koutodoor.A, Adrotator.A, Startpage, Kerproc.RTS.

The lower accuracy in certain families arises mainly due to two reasons. The first is due to the visual dissimilarity of images in families such as Orsam.RTS, Kerproc.RTS. This dissimilarity could be due to a disparity of the labeling scheme (unlike our other experiments, our labels here are derived from one AV source). The second reason is because some families such as Startpage were classified as Starpage.DE or Startpage.E, which presumably imply that are variants derived from the same family. This is not completely incorrect since there is only a misclassification in the subscript and not on the family name. However, for completeness we still present these discrepancies as misclassification in our results, as we treat Microsoft's labels Startpage, Startpage.DE, and Startpage.E as three separate families in our analysis.

Anubis Dataset: We also performed image similarity analysis on 685,490 malware binaries which we obtained from the authors of [27]. These malware were further clustered into 1,441 behavioral clusters using the clustering technique proposed in [27] and we were given the cluster labels. We used these labels as the ground truth and performed k-NN based supervised classification with 10-fold cross validation to obtain a classification accuracy of 71.8%. In other words, close to 500,000 malware inside the behavioral clusters are visually similar. This further reinforces that image similarity analysis is comparable with dynamic analysis even on a large malware corpus.

Offensive Computing Dataset: We finally repeated the experiments on a malware corpus of 1.4 million malware which we obtained from Offensive Computing LLC and

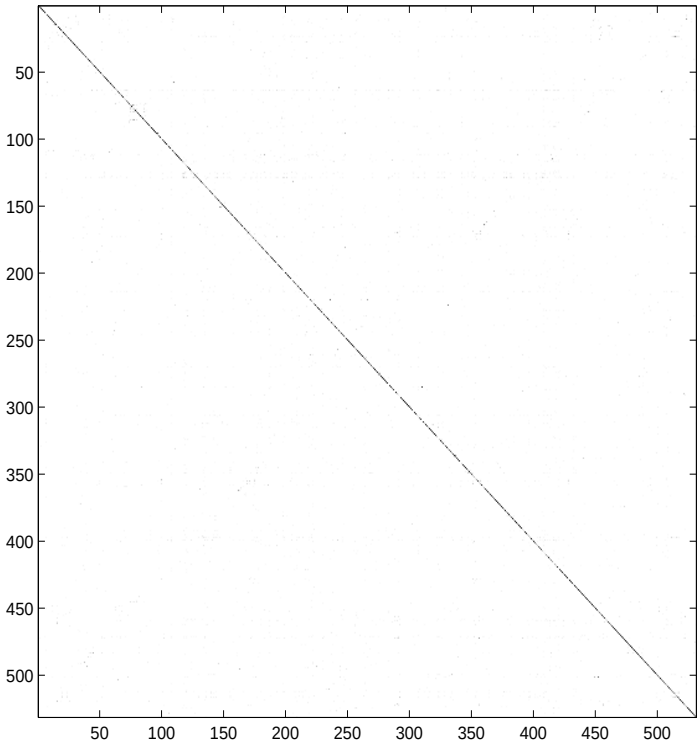


Figure 3.25: Large scale analysis with confusion matrix for 63,002 malware comprising 531 families

we obtained a classification accuracy of 78.2%. This shows that close to 1.1 million malware were classified correctly.

3.6 Summary

In this chapter we have proposed a novel approach to analyze malware by representing malware binaries as digital images and using image similarity descriptors to compactly summarize malware. With extensive experiments, we showed how this method

can be used for malware similarity and classification. Our method scales well and is comparable in performance with state of the art static and dynamic analysis approaches but several times faster. This technique could be a useful complement to current malware analysis strategies. In the next chapter we will describe two systems based on this approach for malware detection and retrieval.

Chapter 4

Malware Detection and Retrieval using Image Similarity

4.1 Introduction

In this chapter we build upon the image similarity approach described in the previous chapter. First we present SigMal, a fast image similarity based malware detection framework. It can operate on both packed and unpacked samples, avoiding the resource intensive unpacking process. We use heuristics based on Portable Executable (PE) structure information and extract image similarity features using a section-aware approach. Our experiments show that SigMal produces the best result in terms of precision, compared to other existing static malware detection methods. We perform large scale experiments on 1.2 Million recent samples, both packed and unpacked, observed over three months and demonstrate that our method can detect 50% of the recent incoming samples with above 99% precision.

In the second part of the chapter (Sec. 4.3) we present SARVAM, a web based system for content-based Search And RetrieVAl of Malware that finds similar malware using image similarity. SARVAM has been operational since May 2012. During this period, we received more than 440,000 samples of which nearly 60% were possible variants of already existing malware from our database. SARVAM lets users to upload malware samples and obtain the possible variants. Our system has been built on a single desktop computer and the average query time is less than 6 seconds on a database of 7 Million malware.

In the third part of this chapter (Sec. 4.4), we discuss some techniques that an adversary who is familiar with our method can use to defeat our approach. The rest of the chapter is organized as follows. In Sec. 4.2 we present SigMal, a framework for fast image similarity based malware detection. We present SARVAM, a web based similar malware retrieval system in Sec. 4.3. In Sec. 4.4 we discuss the adversarial ways to defeat our approach. The summary is presented in Sec. 4.5.

4.2 SigMal: SIGnal processing based MALware triage

4.2.1 Section-aware feature extraction

A normal executable file structure consists of many sections, such as code or data. Apart from some special types of malware executables, such as COM files, usually all malware executable files are also structured in this way. The true malicious behavior of malware is represented by the section containing the code, which executes the actual malicious activities. Computing GIST on the entire image ignores this critical information and generates malware fingerprints that are agnostic to the internal executable

structure. This approach may cause the code section similarity to be out-weighted by dissimilarities of other sections, such as resource section, and fail to identify a variant. Generic packers and installers usually share resources, such as icons and extraction routines. With relatively small packed executables, these resource similarities will produce false-positive similarity detection.

SigMal takes advantage of the internal structure of an executable. The texture properties of an executable section depends on its content type. An executable can contain different types of contents such as code, packed and unpacked data, and other resources, which produce corresponding different types of GIST filter responses. The texture feature extracted from the entire binary captures the spatial structure of these contents. To capture more localized information from the important regions of the binary, we extract separate feature sets from each "important" section of the executable. We say a section is an important section if it is likely to contain the code (packed or unpacked) of the executable. These sections get more weight in the feature set, because the texture features are computed on a per-section basis and represented separately in the feature vector. To extract separate feature sets from the important sections of the binary, we first need to identify them. One way to extract this information is to use the PE file structure information. However, especially in case of malware binaries, the mapping of section information from the PE structure to the binary file data is not always reliable. For example, code sections can be compressed, relocated, or obfuscated, and their size can be spuriously specified as an arbitrarily large value. We use heuristics to find boundaries of the important sections of an executable within the raw binary data, and select two important sections using the heuristics presented in Algorithm 1. These heuristics are loosely based on [52]. We also experimented with heuristics that priori-

Algorithm 1 Finding important sections

Data: PE Executable

Result: A list of important sections

```
Map sections into raw binary file  if overlapping section exits then
    resize section to make it contiguous with adjacent sections
end
if .text section exists then
    if is the largest section then
        Result.append(.text section and the second largest section)
    else
        if .text section is writable then
            Result.append(two largest sections)
        else
            Result.append(.text section and the largest section)
        end
    end
else
    if  any non-writable executable section exists then
        Result.append(this section and the largest section)
    else
        Result.append(two largest sections)
    end
end
```

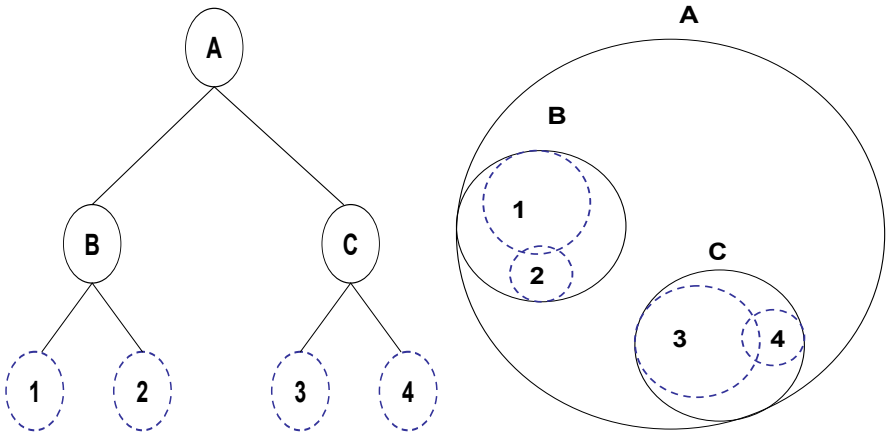
tizes the section selection based on entropy instead of size. However, the results were slightly less precise.

4.2.2 Methodology

Our approach to malware detection is based on instance-based learning. That is, the model learns from instances of known samples. To classify an unknown sample, the model finds the most similar instance and returns its class label as the prediction. A variation of this approach is the k-*nearest-neighbor* method, where class label is deduced from the top-k most similar instances by majority vote. This method with a larger value of k is suitable where enough learning instances are available to form a majority vote for each class. In our case, $k = 1$ produced the best detection results.

Feature matching

To find the nearest-neighbor sample in the learning dataset, we use the Euclidean distance metrics between feature vectors. For each unknown sample q, we perform nearest-neighbor query on the malicious and the benign datasets, which returns two distances, say d_m and d_b, respectively. If both distances are very similar, we cannot say for sure whether the sample q is malicious or benign. This confusion is even more pronounced when d_m and d_b themselves are large (i.e., when the similarity to the dataset is weak). In order to model this nature of the distances, we introduce a detection confidence parameter c as described in Eq. 4.1. We mark the sample q as *unknown* if the value of c is less than certain threshold (i.e., the absolute difference of distances d_m and d_b is not large enough).



(a) Binary Tree (b) Corresponding Ball Tree

Figure 4.1: Illustration of a Balltree

$$c = \frac{|d_m - d_b|}{\sqrt{d_m^2 + d_b^2}} \tag{4.1}$$

Here, the value of detection confidence c varies from 0 to 1, such that the value of 0 implies no confidence (unknown), and the value of 1 implies the highest confidence. In case of a faulty training set, where the same sample is present in the both malware and benign datasets, the value of c for that sample will be undefined ($d_m = d_b = 0$).

Fast nearest-neighbor

Because of the high dimensionality of the feature SigMal, brute-force nearest-neighbor search is computationally expensive. For the efficient nearest-neighbor search in a high-dimensional space, we use *Balltree* data structures [78]. A Ball, in n-dimensional Euclidean space R^n, is defined as a region bounded by a hyper sphere. It is represented as $B = \{\underline{c}, r\}$, where $\underline{c}$ is an n-dimensional vector specifying the coordinates of the

ball's centroid, and r is the radius of the ball. A balltree is a binary tree where each node is associated with a ball. Each ball is a minimal ball that contains all balls associated with its children nodes. The data is recursively partitioned into nodes defined by the centroid and the radius of the ball. Each point in the node lies within this region. Fig. 4.1 shows an illustration of a binary tree, and a Balltree over four balls (1,2,3,4). Search is carried out by finding the minimal ball that completely contains all its children. This ball also overlaps the least with other balls in the tree. For a dataset of M samples and dimensionality N, the query time grows approximately as $O[N \log(M)]$ (as opposed to $O[NM]$ for a brute force search).

We conduct a small experiment to compare the query time and build time. We choose 500 pseudorandom vectors of dimension 320 (same as GIST). These are sent as queries to a larger pseudorandom feature matrix of varying sample sizes (from 100,000 to 2 Million) and same dimension. The total build time and total query time are computed for the cases of brute force search and Balltree-based search (Fig. 4.2). We see that there is a significant difference in the query time between the Balltree-based search and brute force search as the number of samples in the feature matrix increases. In the case of build time, the time taken to build a Balltree increases as the sample size increases. In practical systems, however, the query time is given more priority than the build time.

4.2.3 Dataset Preparation

Benign dataset: For the first dataset, we collected benign executables from three different sources, a fresh Windows XP SP2 install, the ZDnet Software Directory, and the National Software Reference Library (NSRL) maintained by NIST. Our benign dataset

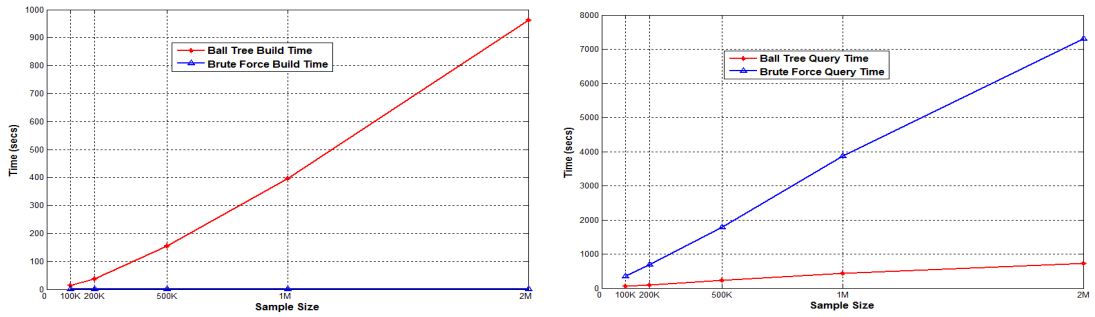


Figure 4.2: Comparison of Brute Force Search and Balltree Search: (a) Build Time (b) Query Time

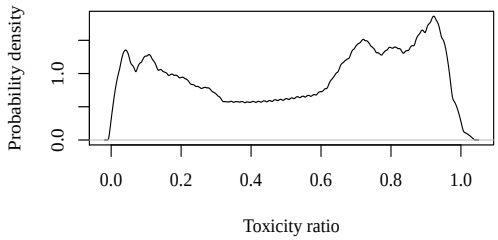


Figure 4.3: The $toxicity\ ratio$ distribution of 1.2 million malware samples.

contains 377 executables from a fresh Windows installation, the top 3000 most popular downloads from ZDnet Software Directory, and 49,373 software binaries from NSRL. We consider the most downloaded software from ZDnet Software Directory as benign. We assume that a malicious software does not appear in the most-downloaded list of a well-reputed site.

Malicious dataset: Malware dataset creation is a difficult problem [65, 87]. We built our second dataset from the executable samples provide to us by the authors of *Anubis* [1]. Anubis is a dynamic malware analysis platform that receives thousands for samples for analysis everyday. We obtained the malware samples submitted to *Anubis* in 2011, along with the latest antivirus labels associated with each sample. Antivirus

labels are provided by VirusTotal [19], which includes labels from different antivirus vendors for each submitted sample. To each sample, we associate a *toxicity ratio* τ, where τ is the ratio of the total number of antivirus vendors that detected the sample as malicious to the total number of antivirus vendors checked by VirusTotal. The density distribution of the *toxicity ratio* of 1.2 million malware samples is shown in Fig. 4.3. It clearly shows that the ratio is concentrated either towards a smaller value or towards a larger value. This means, either only a few anitvirus vendors are likely to label a sample as malicious (sometimes spurious) or almost all vendors are likely to label it as malicious. From this set of samples, we built the malicious dataset by taking samples with $\tau > 0.9$, that is, the set of binary samples which were flagged as malicious by 90% of the antivirus vendors. To maintain an effective large majority, we discarded those samples which have results from less than 30 antivirus vendors (out of 49). This consensus by a majority of antivirus vendors is a result of many human experts who have analyzed the sample (or similar samples) and concluded it to be malicious. Moreover, to have a stronger confidence on this consensus, we chose older samples observed in 2011 with their latest antivirus labels obtained after a year.

The samples-per-malware-family metric of this type of datasets are usually skewed because of the abundance of some widely popular malware families, which are usually more frequently submitted to such public analysis platform, such as Anubis. Therefore, out of the large dataset observed over three months, we only took at most 100 samples per malware family. The dataset after this selection contains 51,058 unique malware samples representing 15,089 unique malware families.

Real-world dataset: To evaluate the performance of the detection methods on real-world malware, we used the recent-feed of malware samples submitted to Anubis over

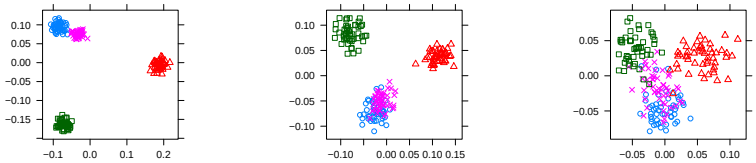


Figure 4.4: Feature robustness against noise. Similar symbols represent the variants from the same seed binary.

three months, starting from November 2012. This dataset contains 1.2 million samples.

4.2.4 Experiment 1: Evaluating SigMal on Malware and Benign Datasets

The image similarity features of both malware and benign datasets are computed and stored in memory in the *Balltree* data structure. In the n-grams-based detection, we used a bitvector approach to encode the n-grams signature, as proposed in [53]. This transforms the Jaccard computation into more CPU-friendly logic operations, and speeds up the computation by many orders of magnitude. Like previous n-grams based works [81, 52], we set n=2 and n=3.

We performed a synthetic experiment to test the robustness of the feature against small modifications (noise) introduced into the sample. We first generated four *seed binaries* containing 200KB random bytes. We chose to use random bytes such that we do not make any specific assumption on the data pattern. From each of these seed binaries, we generated synthetic variants by introducing random noise to the original binary. Note that the differences among the variants are even more pronounced due to this random modifications. We computed image similarity features from these synthetic variants and visualized using Multidimensional scaling (MDS), as shown in Fig. 4.4.

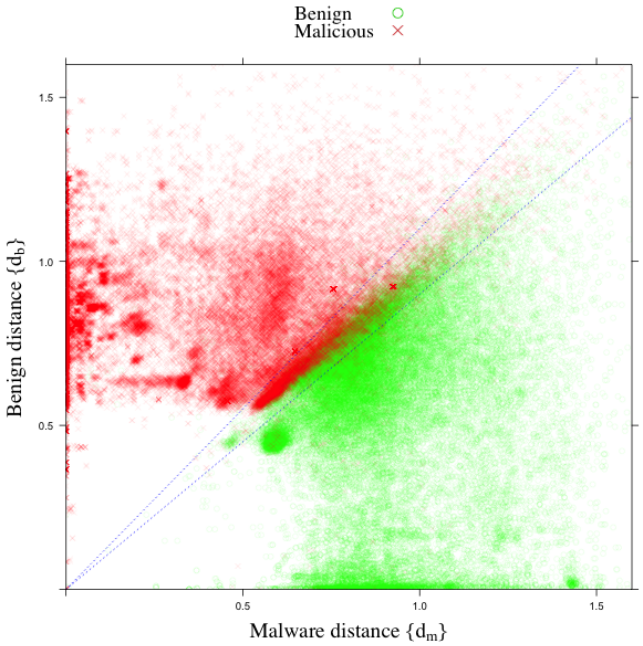


Figure 4.5: The nearest-neighbor distance distribution of 100K samples from the 10-fold cross-validation experiment of SigMal. For each point, X-axis represents the nearest-neighbor distance to malware dataset, and Y-axis represents the the nearest-neighbor distance to the benign dataset.

We can see that the features can be used to cluster the variants even when 50% of the original bytes are randomly modified. In the case of malware binaries, such noise may be introduced by polymorphic or metamorphic engines.

Detection: At first, we analyzed the classification strength of the SigMal features using various machine learning classifiers: Nearest Neighbors (NN), k-Nearest Neighbors (k-NN), Support Vector Machine (SVM, with radial basis kernel), Decision Trees, Naive Bayes and Random Forest Classifier. We observed that the Nearest Neighbor (NN) classifier outperforms all other machine learning classifiers. Best results were produced when single nearest-neighbor distances were used for the detection (k=1). Increases in

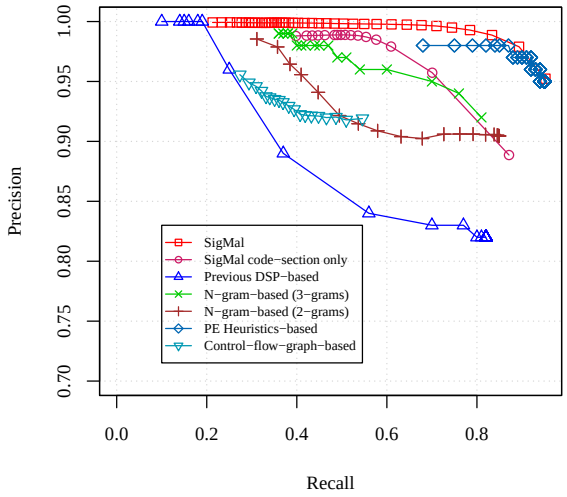


Figure 4.6: Comparison of malware detection algorithms.

the value of k slightly degraded the performance of the detection. In the next step, using Nearest Neighbor (NN) classifier, performed a comparative evaluation of SigMal with existing detection methods. We used the standard 10-fold cross-validation process on the same dataset for all methods. The evaluation dataset used in this experiment is described in Sec. 4.2.3. We used Precision-Recall(PR) analysis instead of Receiver Operating Characteristic (ROC) analysis for comparing performance of different detection methods. When dealing with skewed datasets, PR analysis gives a more informative picture of an algorithm's performance [41].

We performed the precision-recall analysis of SigMal by varying the threshold value t of the detection confidence parameter c (introduced in Sec. 4.2.2). For each testing sample, SigMal returns two nearest-neighbor distances d_m and d_b corresponding to the malware and benign training sets, respectively. Fig. 4.5 shows the distribution of these distances for all samples as computed from the 10-fold cross-validation experiment.

A sample with shorter distance to the malware dataset than to the benign dataset falls in the upper left section of the graph. The area inside the dotted line represents the confusion area given by the inequality $c < t$, when the threshold value is chosen as $t = 0.1$. Samples within this confusion area are marked as *unknown* because their detection confidence value (c) is not large enough. A change in the threshold value t changes the performance of the detection. Higher values of threshold t produces more precise results by widening the confusion area, while reducing the recall rate.

Comparison with Other Features: We compared our image similarity similarity algorithm with three popular malware detection methods: n-grams, control flow graphs (CFG) and file structure based features. To compare n-grams, we used the Jaccard similarity metric as previously explained in Sec. 2.1.3. For CFG comparison, we use the technique followed in [61], which decomposes the CFG into a set of subgraphs and the maximum common subgraphs between two CFGs measures the similarity (Sec. 2.1.3). For file structure based features, seven PE based features are extracted and J48 decision tree method to build models for the malware and benign classes.

Both n-grams based method and CFG based method provide similarity measures instead of distance measures. To perform the precision-recall analysis, we vary a threshold s, which, in this case, is a threshold for the minimum similarity. If a resulting value of the similarity measure for a query sample is less than the threshold parameter s, we consider the value to be too weak, and the sample is marked as *unknown*. In case of the file-structure-based detection, we vary the output class probability of the decision tree classifier.

The precision-recall analysis of all the detection methods is presented in Fig. 4.6. One can see that our method has the best overall performance. We achieved very high

precision (99.66%), while still maintaining a good recall rate of about 50%. When only the code-section of the executable is used to extract the features, the performance of the algorithm degrades. This shows that statistical similarity of executables only based on the code-section is weaker. As reported in previous works [93, 95], PE-structure-based methods produced overall good precision and recall rates. However, it could not improve the precision above 98%. In cases where greater recall is important, our method still has second best precision compared to the rest of the methods. n-grams based method (when $n = 3$) has relatively good overall performance. However, the computational overhead is high. Since the packed samples were not unpacked, as expected, the control-flow-graph-based method did not produce good results. We are aware that for better results, CFG requires a packed sample to be unpacked, which is a hard problem in malware analysis. However, the scope of this experiment is to perform a comparison of the detection methods when faced with unmodified samples found in the wild.

Performance: In this section we compare the time and space efficiency of the algorithms. We focus on the time required by two main steps: building the features, and detecting the similarity. We measured the memory requirements to compare the space efficiency. All of the feature-extraction processes were modified from multi-threaded implementations to single-threaded implementations for this particular experiment. All performance experiments were done in the same computational environment (Linux 3.2.0-35 machine, Intel i7 3.33GHz/12GB).

We measured the computation time required by the feature extraction of ten thousands samples for each algorithm. The average time required to compute these different types of features are presented in Tab. 4.1. Since the PE-heuristics method inspects only

the header part of the executable, it is the fastest among all. SigMal feature extraction is about five times faster than the CFG feature extraction, and seven times faster than the N-gram feature extraction. We also measured the average per-sample space requirements for storing raw bytes of the features in the memory. Again, the PE-heuristics method requires the least amount of space to store features, since its feature dimensionality is the smallest. SigMal features require two times less memory compared to n-grams features (2-grams), and 78 times less memory compared to CFG features.

Table 4.1: Average per sample feature extraction time in seconds and per sample memory requirements in KBs

	$SigMal$	$N\text{-}gram$	$PE\text{-}heuristics$	CFG
$Time$	0.0265	0.1965	0.0024	0.1379
$Space$	3.783	8.000	0.0664	297.745

To evaluate the scalability of the algorithms, we performed the 10-fold cross-validation experiments using datasets of increasing number of samples, such that both the training samples and the testing samples are increasing in each experiment. We chose this option to resemble a real-world scenario, where both the number of new known malware samples (training set) and the number of samples to be inspected (testing set) are continuously increasing. For each 10-fold experiment, we computed the average of the total time required for the detection query in each fold. The results are presented in the Fig. 4.7. We can see that our method is easily scalable to hundreds of thousands of samples. In a 10-fold cross-validation experiment on a dataset of 100,000 samples (80,000 in the training set, 20,000 in the testing set) average per sample query response was 47.95 milliseconds. The quadratic increase in the query response time of N-gram-based and CFG-based approaches is primarily because of the $O(n * m)$ computational

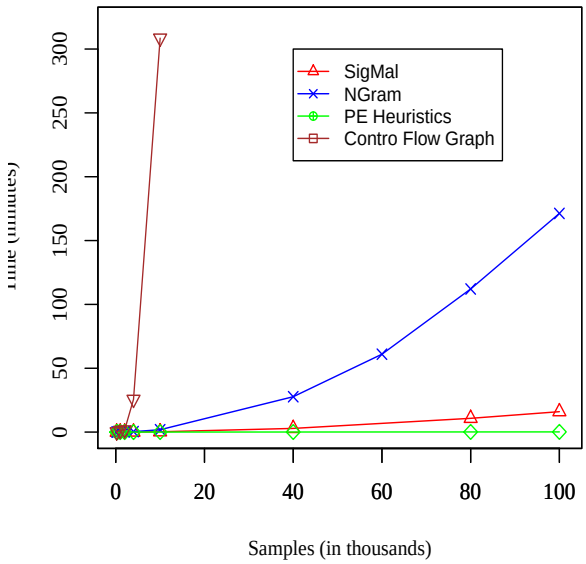


Figure 4.7: Query performance comparison. X-axis is the number samples used in each 10-fold experiment. Y-axis is the average of the total query time in each fold of that experiment

cost of the Jaccard-similarity comparisons. As expected, the decision tree-based PE-heuristics method is the fastest among all.

4.2.5 Experiment 2: Evaluating SigMal on Real-World Data

We showed that our method works well in a limited dataset of old samples. Many of the previous works were also evaluated using a similar dataset. However, we wanted to evaluate their performance when applied to a large dataset of recent real-world samples. Results from such datasets can demonstrate the true applicability of a method.

We observed that when an old malware dataset was used as the ground truth for the detection of recent malware, as expected, the detection precision was poor. When we took the entire malware dataset as the ground truth, the query response time of SigMal significantly increased. This is because of the increased search-space for the

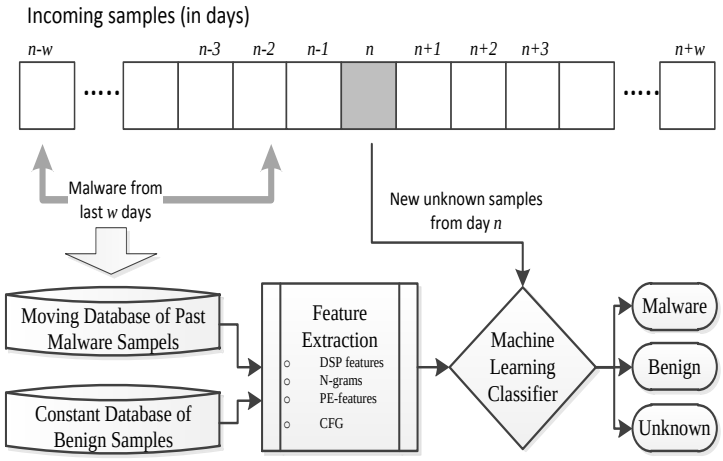


Figure 4.8: Overview of the sliding window experiment on the real world samples

nearest-neighbor query. The scalability problems with other methods were even more critical while using larger ground truth datasets. Because of this, we prepared a fresh malware ground-truth everyday using the malware samples observed in the recent past. The overview of this approach is presented in Fig. 4.8. More precisely, for the n^{th} day's experiment, we build a dynamic malware dataset by taking the set of malware samples observed in the past time window of w days (from day $(n - w)$ to day $(n - 1)$). We can infer from the *toxicity* density distribution at Fig. 4.3 that the confidence builds up around $\tau = 0.6$. Hence, for the recent samples, we used a less conservative value of $\tau > 0.6$ for building the dynamic training set. We use the samples submitted on day n as the testing set for the n^{th} day experiment. For the comparison of the SigMal detection results, we need to label each of these incoming samples as benign or malicious using recent antivirus labels. However, we noticed that few antivirus vendors falsely detect a benign sample as malicious, if packed with some commonly available executable packer, such as Winpack. To avoid including such spurious labels, we consider $\tau <=$

0.3 as a low confidence value for the result evaluation, and exclude it in our precision-recall analysis.

To find the optimal sliding time window in terms of speed and accuracy for collecting the ground truth, we performed a set of experiments using different values of w, ranging from 7 days to 60 days. The precision did not improve significantly when the time window was greater than 30 days. Therefore, we chose the time window w as 30 days for the rest of our daily experiments.

As seen in the performance experiments in Sec. 4.2.4, n-grams based and CFG based methods have a high computational cost. For example, with the 30 days sliding-window dataset, the CFG based method required several days to complete a single sliding window experiment even with a parallelized implementation on a 24-core 96GB machine. Hence, this part of the comparison experiment on the real-world dataset is limited to few days.

Results: The precision and recall performance of the sliding window experiments are presented in Fig. 4.9. We can see that at $t = 0.33$, more than 50% of the recent daily samples can be accurately detected as malicious or benign with a precision of 99.5% (standard error 0.000835). This can essentially reduce about 50% of the resource requirements of a triage system by avoiding further, more expensive, analysis.

Fig. 4.10 presents the detection results of all methods, when applied to the sliding window dataset (i.e., samples from the month of November 2012 as the training set, and the samples from December 1st 2012 as the testing set). One can see that the performance of other methods is less impressive in terms of precision. None of the methods has a high precision result. The PE-heuristics-based method and CFG-based method have a relatively larger recall. However, they do not produce high precision

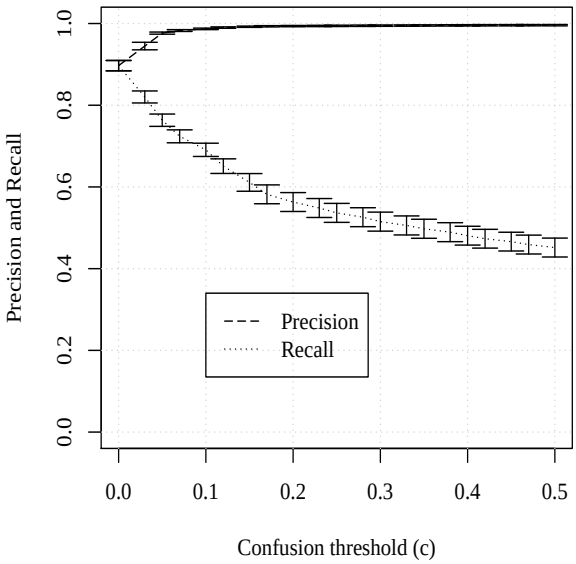


Figure 4.9: Precision and Recall of the SigMal detection on the real world samples observed by Anubis in December 2012 and January 2013. The figure represents the mean values of the daily results and the standard error.

results. In case of the CFG-based method, its computation cost is yet another critical factor that makes it unsuitable for large-scale malware triage.

4.2.6 Experiment 3: Evaluating SigMal on current AV labels

Notice that the detection results were checked with the VirusTotal results obtained at the time of the submission. However, there is a possibility that some malware may not have been detected by the majority of antivirus vendors at that time. Antivirus vendors may not eventually detect all malware. Hence, an accurate analysis of such false-positive is difficult. Here, we are only interested in how many malware samples SigMal could have detected that AV vendors missed at the time of submission, but later identified them as malware. We used an old dataset observed in 2011. For this dataset, we have old antivirus labels retrieved from VirusTotal during the time of the submis-

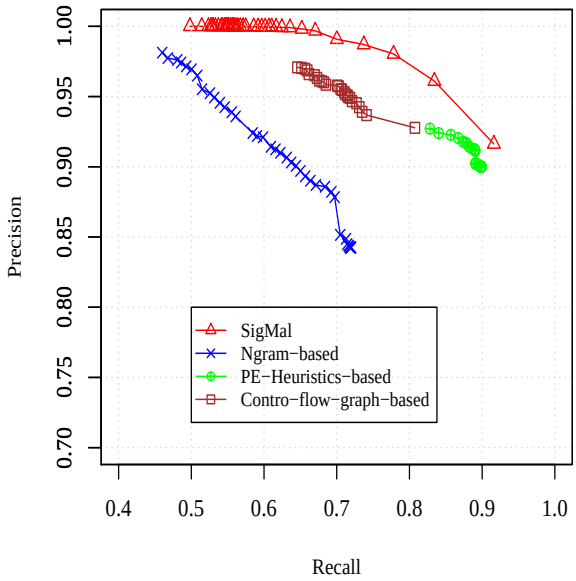


Figure 4.10: Comparison of malware detection methods with live malware feed (2012-12-01)

sion. We re-submitted the old samples to VirusTotal for rescanning and retrieved the latest antivirus labels. We performed a simulated daily sliding-window experiment on this dataset and re-evaluated our results with the updated antivirus labels. We found out that SigMal could have detected, on average, 70 malware samples per day before any antivirus vendor detected them as malicious. We also performed a similar experiment with recent samples, however, waiting for a month only before retrieving the new labels. We found 210 cases of malware in the month of December 2012 that SigMal could have detected, while none of the antivirus vendors flagged them as malicious at the time of the submission.

Packed Benign Samples: There is a possibility that the classifier may be biased on detecting packed and unpacked executables, instead of malicious and benign. In order to evaluate this, we performed 10-fold cross-validation experiment using packed benign samples. We first packed the samples from the benign dataset using three popular exe-

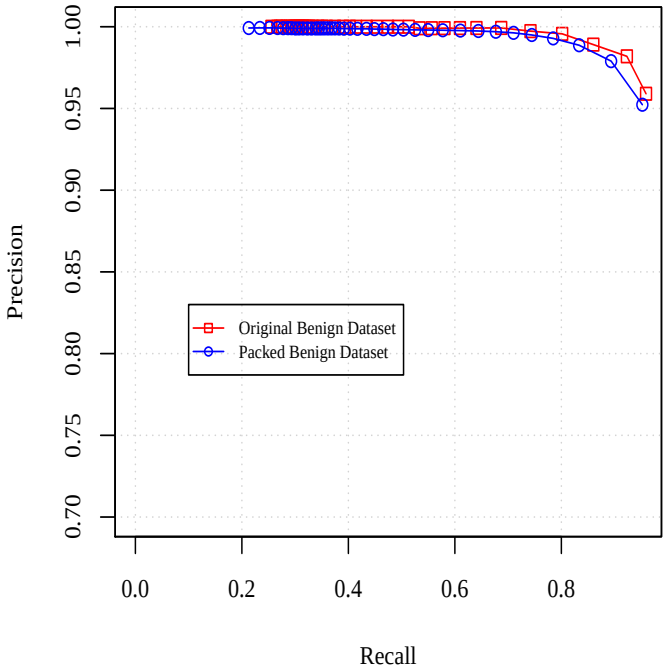


Figure 4.11: Precision-recall comparison with packed-benign dataset

cutable packers: UPX, WinUpack, and NSPack. We built the benign dataset from these packed samples. We used the same malware dataset from 2011 for this experiment. The 10-fold cross-validation result in Fig. 4.11 shows that the result is comparable to the experiment with normal benign samples.

4.3 SARVAM: Search And RetrieVAl of Malware

In this section we utilize signature extraction techniques from image processing and build a system, SARVAM, for large scale malware search and retrieval. Leveraging on past work in finding similar malware based on image similarity [72], we use these compact features for content-based search and retrieval of malware. These features are known to be robust, highly scalable and perform well in identifying similar images in a web-scale dataset of natural images (110 million) [43]. They are fast to compute and are shown to be 4,000 times faster than dynamic analysis while having similar performance in malware classification [76] and also used in malware detection [55]. These image similarity features are computed on a large dataset of malware (more than 7 million samples) and stored in a database. For fast search and retrieval, we use a scalable *Balltree*-based Nearest Neighbor searching technique. This reduces the average query time to 6 seconds for a given query. We built SARVAM as a public web-based query system, (accessible at *http://sarvam.ece.ucsb.edu*), where users can upload queries and obtain similar matches for that query. Fig. 4.12 shows the web interface of SARVAM. The system has been active since May 2012 and we have received more than 440,000 samples since then. For a large portion of the uploaded samples (approximately 60%), we were able to find variants in our database of 7.1 Million samples.

Other similar systems that let users upload malware include Virustotal [19], Anubis [1] and Malwr [9]. However, while the above systems do static and/or dynamic analysis on a malware sample, ours is the only existing system that finds similar malware. Other related systems from the context of malware similarity and information retrieval include VILO [62] and NEO [92]. We give an overview of SARVAM below. The results are shown on samples uploaded between May 2012 and December 2013.

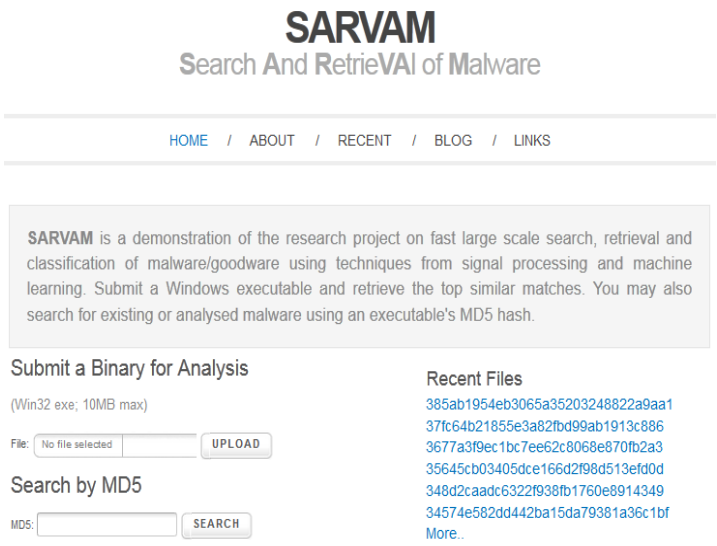


Figure 4.12: Web Interface of SARVAM

At that time, our database had around 4 Million samples and we had received more than 212,000 samples. The average query time was 3 seconds.

4.3.1 Overview of SARVAM

A content-based search and retrieval system is one in which the content of a query object is used to find similar objects in a larger database. Such systems are common in the retrieval of multimedia objects such as images, audio and video. The objects are usually represented as compact descriptors or fingerprints based on the their content [90].

SARVAM uses image similarity fingerprints to compactly describe a malware. These effectively capture the visual (structural) similarity between malware variants and are used for search and retrieval. There are two phases in the system design as shown in Fig. 4.13. During the initial phase, we first obtain a large corpus of malware samples from various sources [1, 12]. The compact fingerprints for all the samples in the corpus

are then computed. To obtain similar malware, we use Nearest Neighbor (NN) method based on the shortest distance between the fingerprints. But the high dimensionality of the fingerprints makes the search slow. In order to perform Nearest Neighbor search quickly and efficiently, we construct a Balltree (explained in Sec. 4.2.2), which significantly reduces the search time. Simultaneously, we obtain the Antivirus (AV) labels for all the samples from Virustotal [19], a public service that maintains a database of AV labels. These labels act as a ground truth and are later used to describe the nature of a sample, i.e., how malicious or benign a sample is. During the query phase, the fingerprint for the new sample is computed and matched with the existing fingerprints in the database to retrieve the top matches. The various blocks of SARVAM are explained in the following sections.

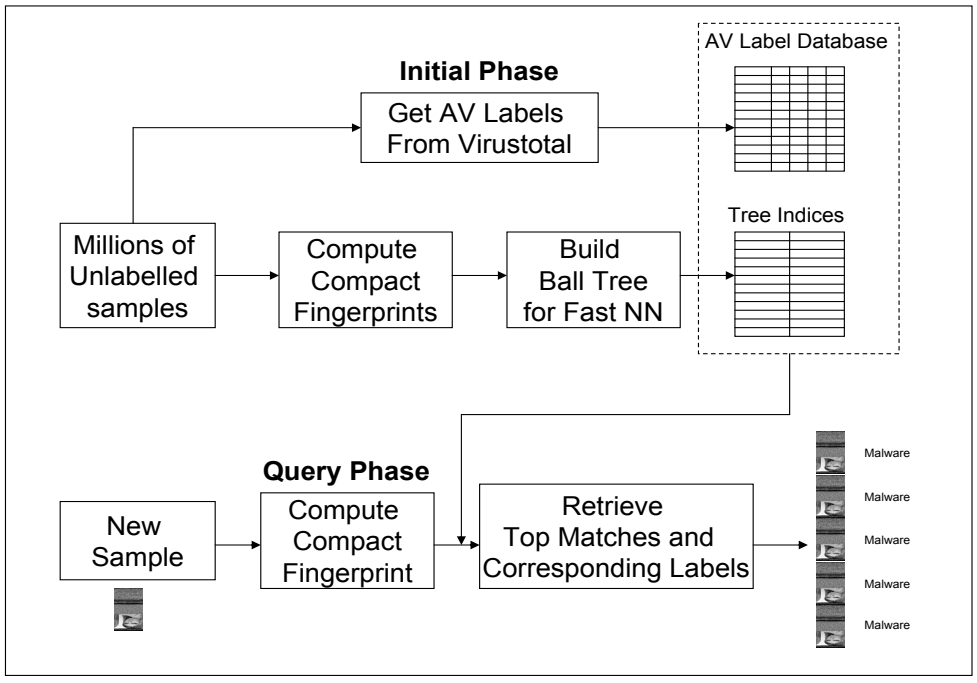


Figure 4.13: Block schematic of SARVAM

4.3.2 SARVAM Implementation

SARVAM is implemented on a desktop computer (DELL Studio XPS 9100 Intel Core i7-930 processor with 8MB L2 Cache, 2.80GHz and a 20 GB RAM) running on Ubuntu 10. The web server is built using Ruby on Rails framework with MySQL backend. Python is used for feature computation and matching. A MySQL database stores information about all the samples such as MD5 hash, file size and number of Antivirus (AV) labels. When a new sample is uploaded, its MD5 hash is updated in the database. All the uploaded samples are stored on disk and saved by their MD5 hash name. A Python script (daemon) checks the database for unprocessed keys and when it finds one, it takes the corresponding MD5 hash and computes the image fingerprint from the stored sample. Then, the top matches for that query are found and the database is updated with their MD5 hashes. A Ruby on Rails script then checks the database and displays the top matches for that sample. The average time taken for all the above steps is approximately 3 seconds.

Database: The SARVAM database consists of approximately 4.3 million samples, most of which are malware. We also include a small set of benign samples from clean installations of various Windows OS. All the samples are uploaded to Virustotal to get Antivirus (AV) labels and these are stored in a MySQL database. Fig. 4.14 shows the distribution of the AV labels for all the samples in our initial corpus. As we can see, most samples have many AV labels associated with them, thus indicating they are most certainly malicious in nature. The corpus and the MySQL database are periodically updated as we get new samples. The AV labels of the samples are also periodically checked with Virustotal and updated if there are changes in the labels. This is because AV vendors sometimes take a while to catch up with the malware and hence, the AV

labels may change.

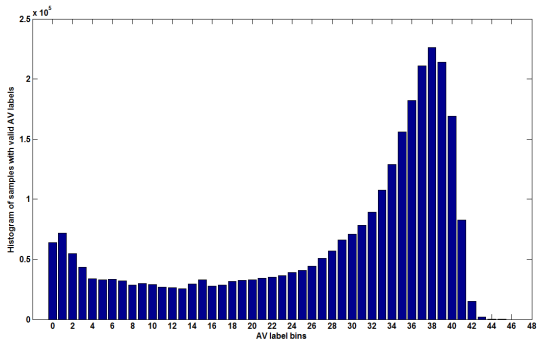


Figure 4.14: Distribution of the number of AV labels in the corpus

SARVAM has a simple web interface built on Ruby on Rails as shown earlier in Fig. 4.12. Some of the basic functionalities are explained below.

Search by upload or MD5 Hash: SARVAM currently supports two ways of search. In the first case, users can upload executables (maximum size 10 MB) and obtain the top matches. In the second, users can search for an MD5 hash and if the hash is found in our database, the top matches are computed. Currently, only Win32 excutables are supported but our method can be easily generalized to include a larger category of data.

Sample Reports: A sample query report is shown in Fig. 4.15. SARVAM supports HTML, XML and JSON versions. While HTML reports aid in visual analysis, XML and JSON reports can be used for script-based analysis.

4.3.3 Design Experiments

For a given query input, the output is a set of matches which are ranked according to some criterion. In our case, the criterion is based on the distance between the query and its top match. We set various thresholds to the distance and give confidence levels to the matches.

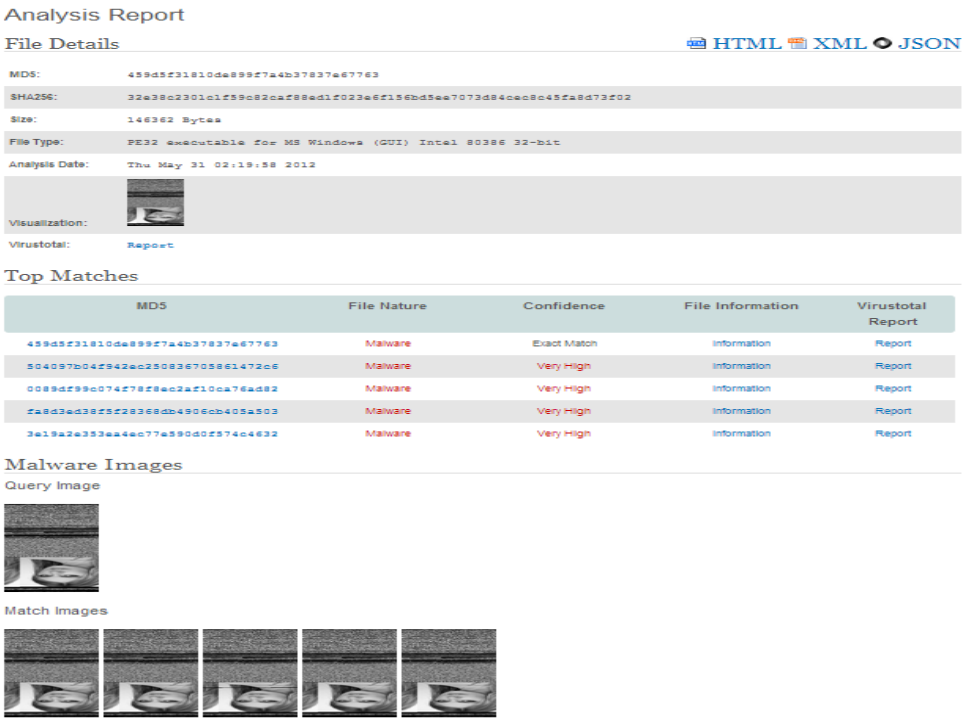


Figure 4.15: Sample HTML report for a query

Design Dataset: Two malware are said to be variants if they show similar behavior upon execution. Although some existing works try to quantify such malware behavior [25, 86], it is not very straightforward and can result in spurious matches. An alternative is to check if the samples have same AV labels. Many works including [25, 86] use AV labels to build the ground truth. We evaluate the match returned for a query based on the number of common AV labels. From our corpus of 4.3 million samples, we select samples for which most AV vendors have some label. In Virustotal, the AV vendor list for a particular sample usually varies between 42 and 45 and in some unique cases goes down to 5. In order to not skew our data, we select samples for which at least 35 (approximately 75% - 80%) AV vendors have valid labels (*None* labels excluded). This resulted in a pruned dataset of 1.4 million samples.

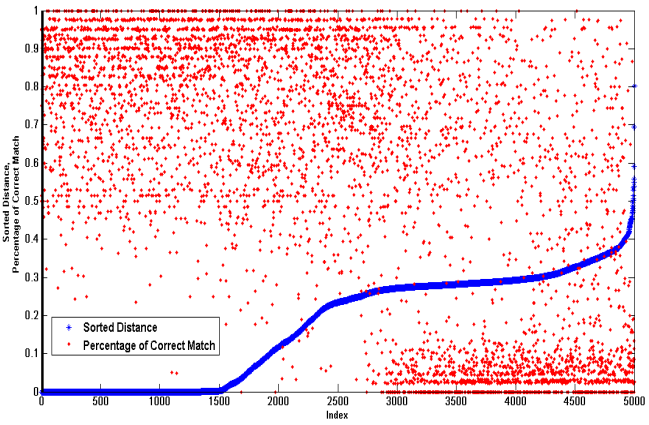


Figure 4.16: Results of Design Experiment on 5000 samples randomly chosen from the training set. Sorted Distance and corresponding percentage of correct match are overlaid on the same graph. A low distance value has a high match percentage while a high distance value has a low match percentage in most cases.

Validation: From the pruned dataset of 1.4 million samples, we randomly choose a reduced set R_s of length $N_{R_s} = 5000$ samples. The remaining samples in the pruned set are referred as the training set T_s. The samples from R_s are queries to the samples in T_s. First, the features for all the samples are computed. For every sample of R_s, the nearest neighbor among the samples of T_s is computed. Let $q_i = R_s(i),\ 1 \leq i \leq N_{R_s}$ be a query, m_i be its Nearest Neighbor (NN) match among the samples in T_s, d_i be the NN distance and AV_{sh} be a set of shared AV vendor keys (such as *Kaspersky, McAfee*). Both the query and the match will have corresponding AV labels (such as *Trojan.Spy, Backdoor.Agent*) for every shared vendor key. We are interested in finding how many matching labels are present between a query and its match, and its relation with the NN distance. The percentage of matching labels pm_i between a query and its match is

defined as:

$$pm_i = \frac{\sum_{j=1}^{N_{AV_{sh}}} I(q_i[AV_{sh}(j)] = m_i[AV_{sh}(j)])}{N_{AV_{sh}}},\ 1 \leq i \leq N_{R_s} \tag{4.2}$$

where $N_{AV_{sh}}$ is the total number of shared AV vendor keys, $q_i[AV_{sh}(j)]$ and $m_i[AV_{sh}(j)]$ are the AV labels of the query and its NN match for the i^{th} query and j^{th} AV vendor key and $I(.)$ is the Indicator function. We are interested in seeing which range of the NN distance d gives a high percentage of best AV label match pm. In order to visualize this, the distances are first sorted in ascending order. The sorted distances and the corresponding percentage of correct match are overlaid in Fig. 4.16. We observe that the percentage of the correct matches are highest for very low distances and they decrease as the distance increases. Our results were the same even we chose various random subsets of 5000 samples. Based on these results, we give qualitative tags and labels for the quantitative results as shown in Tab. 4.2.

Qualitative vs Quantitative Tags: For every query, we have the distance from its nearest neighbor and can compute the percentage of correct match between their labels. In reality, only the AV labels of the nearest neighbor are known and AV labels of the query may not be available. Hence, based on the NN distance and the number of AV labels that are present in a match, we give qualitative tags.

Intuitively, we would expect that a low distance would give the best match. A low distance means the match is very similar to the query and we give it a tag of *Very High Confidence*. As the distance increases, we give qualitative tags: *High Confidence*, *Low Confidence* and *Very Low Confidence* as shown in Tab. 4.2.

Very High Confidence Match: A Very High Confidence match usually means that the query and the match are more or less the same. They just differ in a few bytes. The example shown in Fig. 4.17 will help illustrate this better. The image in

Table 4.2: Confidence of a Match

Distance d	Confidence Level	Percentage of pm	Median of pm	Mean of pm	Std. Deviation of pm
< 0.1	Very High	38.6	0.8462	0.7901	0.1782
(0.1,0.25]	High	15.24	0.7895	0.7492	0.2095
(0.25 ,0.4]	Low	44.46	0.1333	0.3454	0.3488
> 0.4	Very Low	1.7	0.0625	0.1184	0.1862

the left is the query image. We see an inverted image of a girl's face which is the icon of the executable. The image in the middle is the top match. If we take a byte by byte difference between the query and the match, we see that most of the bytes in the difference image is zero. Only 323 bytes out of 146304 bytes (0.22%) are non-zero. The distance of the match from the query will usually be lesser than 0.1.

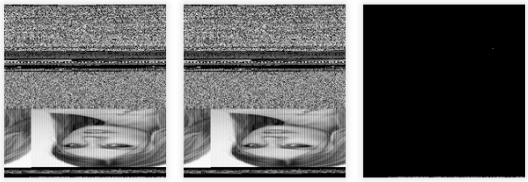


Figure 4.17: Example of a very high confidence match. The image in the left is of the query while in the middle image is of the top match. Shown in the right is the difference between the two. Only a few bytes in the difference image are non-zero.

High Confidence Match: When we talk about a high confidence match, most parts of the query and the top match are the same but there is a small portion that is different. In Fig. 4.18, we can see the image of the input query in the left. The image of the top match in the middle appears visually similar to the query. But the difference image shows that 11,108 out of 80,128 non-zero values (13.86%). Most variants in this category are usually packed variants which have different decryption keys. The distance between the query and the top match will usually be between 0.1 and 0.25.

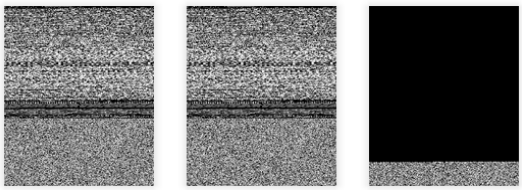


Figure 4.18: Example of a high confidence match. The image in the left is of the query while in the middle image is of the top match. Shown in the right is the difference between the two. A small portion in the difference image are non-zero.

Low Confidence Match: For low confidence matches, a major portion of the query and the top match are different. We may not see any visual difference between the input query and the top match but the difference image clearly shows the huge difference in bytes (Fig. 4.19). These would usually be packed variants (UPX in this case). In the difference image, 75,353 out of 98304 non zero (76.6%). The distance is usually greater than 0.25 and less than 0.4. Low Confidence matches also end up in False Positives (meaning the top match may not be a variant of the query) and hence they are tagged as *Low Confidence*. In these cases, it is better to visually analyze the query and the top match before arriving at a conclusion.

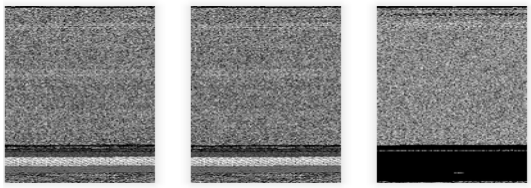


Figure 4.19: Example of a low confidence match. The image in the left is of the query while in the middle image is of the top match. Shown in the right is the difference between the two. A significant portion in the difference image are non-zero.

Very Low Confidence Match: For matches with Very Low confidence, in most of the cases the results don't really match the query. These are cases where the distance is

greater than 0.4.

Apart from the confidence level, we also give qualitative tags to every sample in our database based on how many Antivirus (AV) labels it has. For this, we obtain the AV labels from Virustotal periodically. We use the count of the number of labels to give a qualitative tag for a sample as shown in Tab. 4.3.

Table 4.3: Nature of a Match

No. of AV Labels	Qualitative Label
0	Benign
[1,10]	Possibly Benign
[11,25]	Possibly Malicious
[26,45]	Malicious
No data	Unknown

4.3.4 Results on Uploaded Samples

Distribution based on Month: From May 2012 onwards till Oct 2013, we received approximately 212,000 samples. In Fig. 4.20, we can see the distribution of the uploaded samples based on the uploaded month. We observe that most of the samples were submitted in Sep. 2012 and Oct. 2012 while the activity was very low in the months of Nov. 2012, Feb. 2013, Mar. 2013 and May 2013.

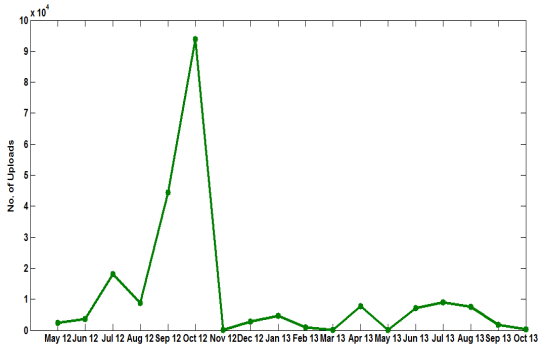


Figure 4.20: Month of Upload

Year of First Seen: In Fig. 4.21, we see the distribution of the year in which the samples were first seen in the wild by Virustotal. Most samples that we received are from 2011, while a few are from 2012 and 2013.

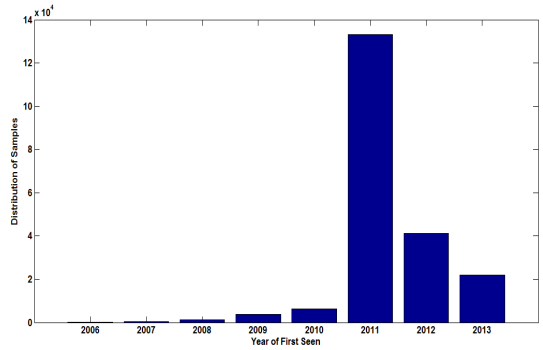


Figure 4.21: Year of First Seen for the Submitted Samples

File Size: The distribution of the file sizes of various samples are shown in Fig. 4.22. We see that most of the files have sizes less than 500 kB.

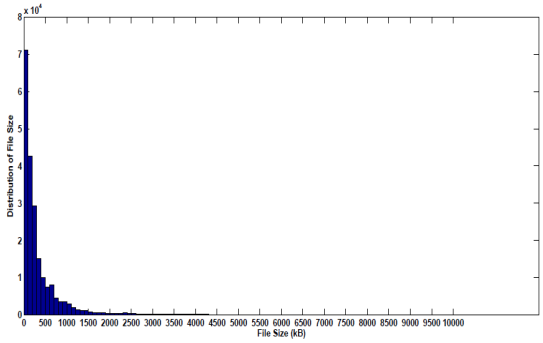


Figure 4.22: File Sizes of Uploaded Samples (kB)

Confidence of Top Match: Of the 212,000 uploaded samples we received, not all the samples have a good match with our corpus database. In Fig. 4.23, we see the distribution of the confidence levels of the top match. Close to 37% fall under Very High Confidence, 8% under High Confidence, 49.5% under Low confidence and 5.5% under Very Low Confidence. This means that nearly 45% of the uploaded samples

(close to 95,400) are possible variants of samples already existing in our database.

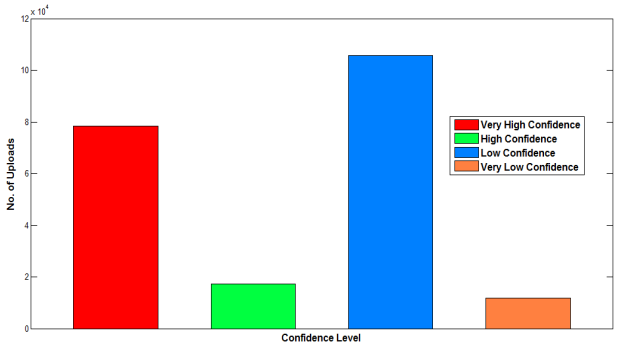


Figure 4.23: Confidence of the Top Match

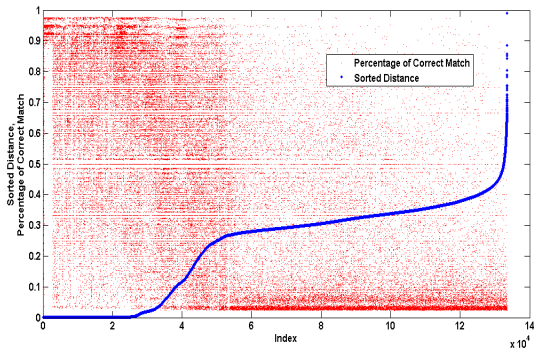


Figure 4.24: For every uploaded sample, the distances of the top match are sorted (marked in blue) and the corresponding percentage of correct match (marked in red) is overlaid.

AV Label Match vs Confidence Level: Here, we further validate our system by comparing the output of our algorithm with the AV labels. For this, we obtained the AV labels for all the uploaded samples and their top match. However, Virustotal has a bandwidth limitation on the total number of samples that can be uploaded. Due to this, we were only able to obtain valid AV labels for a subset of the uploaded samples. We also exclude the uploaded samples that were already present in our database. The percentage of correct match is then computed. Fig. 4.24 shows the sorted histogram

of the distance between the uploaded samples and their top match. Similar to the results obtained in our earlier design experiment (Fig. 4.16), we see that the percentage of correct match is high for a low distance. In Fig. 4.25, we plot this distance versus the percentage of correct match and see that the trend is similar. However, there are a few cases which have a low percentage of correct match for a low distance. This is because we do a one-one comparison of AV labels and malware variants may sometime have different AV labels. For example, the variants in Fig. 4.17, have AV labels *Trojan.Win32.Refroso.depy* and *Trojan.Win32.Refroso.deqg* as labeled by *Kaspersky* AV vendor. Although, these labels differ only in a character, we do not consider this in our current analysis and these could result in a low percentage of correct match despite having a low distance.

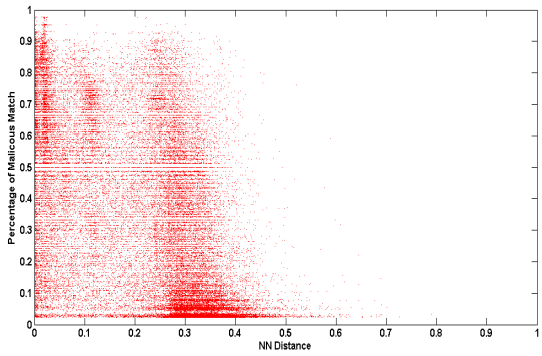


Figure 4.25: Distance vs Percentage of Correct Match

Confidence vs Year of First Seen: For all the uploaded samples, we obtain the year that it was first seen in the wild from Virustotal and compare it with the Nearest Neighbor (NN) distance d. In Fig. 4.26, we plot the year of first seen and the NN distance. We observe that most samples were first seen in the wild between the years 2010 and 2013. Many samples from 2012 and 2013 have a low NN distance and this shows that our system has good matches even with most recent malware. If we consider

only the very high confidence and high confidence matches and analyze their year of first seen(Fig. 4.27), we observe that a large number of samples are from 2011 and a reasonable amount are from 2012 and 2013.

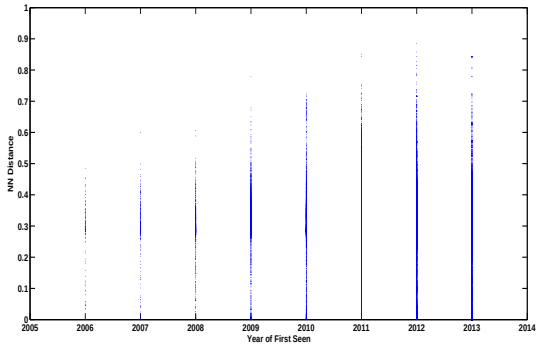


Figure 4.26: Year of First Seen vs Distance

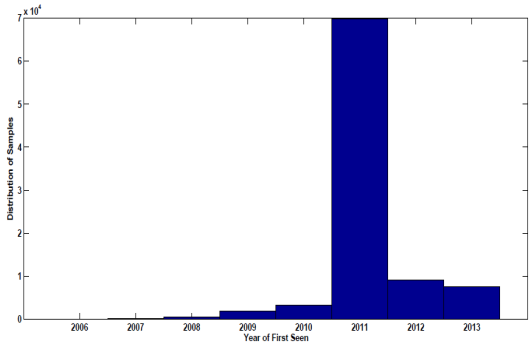


Figure 4.27: Year of First Seen for Very High Confidence and High Confidence Matches

Packed Samples: We also analyze the performance of SARVAM on packed malware samples. One problem that arises here is that identifying whether an executable is packed or not is not easy. In this analysis, we use the packer identifiers *f-prot* and *peid* that are available from the Virustotal reports. Only 39,260 samples had valid *f-prot* packer signatures and 49,333 samples had valid *peid* signatures. The actual number of packed samples is usually more but we consider only these samples in our anal-

ysis. Of these, 16,055 samples were common between the two and there were 970 unique *f-prot* signatures and 275 unique *peid* signatures in the two sets. This shows the variation in the signatures of these two packer identifiers. For both cases, the most common signature was *UPX*. Others included *Aspack*, *Armadillo*, *Bobsoft Mini Delphi* and *Pecompact*. The signature *BobSoft Mini Delphi* need not always correspond to a packed sample and it could just mean that the sample was compiled using *Delphi* compiler. For both sets of samples, we obtain their NN distance and plot the sorted distance in Fig. 4.28. We observe that nearly half the samples in both sets fall in the Very High Confidence and High Confidence range.

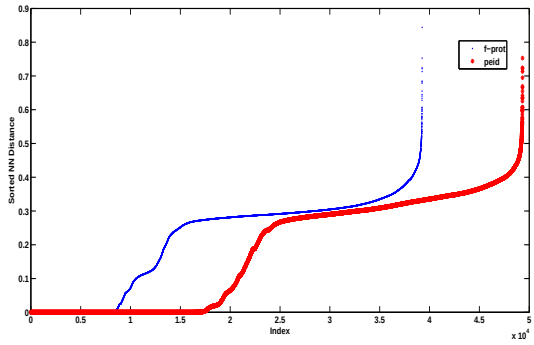


Figure 4.28: Sorted NN Distance of Packed Samples

Next, we consider only packed samples that fall in the Very high Confidence and High Confidence range (NN distance $d <= 0.25$). This reduced the samples identified by *f-prot* to 14,936 and *peid* to 24,098. Tab. 4.4 shows the top 5 packer signatures of *f-prot*, with the total number of unique signatures being 364. *UPX* was the most common signature while others included *Allaple*, *PECompact* and *Aspack*. In the case of *peid*, the number of unique signatures was 186. The top 5 are shown in Tab. 4.5. Again, *UPX* was the most common followed by *Armadillo*, *Bobsoft Mini Delphi* and *PECompact*. This analysis further shows that our approach works on different types of

packed samples as well.

Table 4.4: Top Packer Signatures of *f-prot*

Packer	No. of samples
UPX	7401
Allaple	920
PecBundle, PECompact	916
Aspack	896
UPX LZMA	724

Table 4.5: Top Packer Signatures of *peid*

Packer	No. of samples
UPX 2.90 [LZMA]	9285
Armadillo v.1.71	4855
BobSoft Mini Delphi	1537
Armadillo v1.xx - v2.xx	1456
PECompact 2.xx	1402

4.4 Adversarial Modeling

In this section we discuss the limitations of our approach. First, the characterization of malware using image based features does not give much information about the actual behavior of the malware. Second, since our approach relies on instance-based learning, its main limitation is that it can only detect/classify malware similar to what has already been observed. Thus, zero day attacks of new unseen malware cannot be prevented. However, this is a generic problem with any similarity based malware analysis framwork. Below, we provide some ways in which an adversary can attack our approach.

4.4.1 Adding blocks of redundant bytes

As a targeted attack to our method, an adversary could insert large unused sections of random data into an executable. This may cause our heuristics to select wrong sections as important sections. Since it is likely that no previous sample matches with the random data, such samples will be considered *unknown*. In a more crafted attack, an adversary could embed code section of a benign executable as its largest section. This will generate the exact same feature vectors corresponding to those sections. However, the malicious code still needs to be embedded in the file to make the crafted executable malicious. Because of this, the part of the feature vector generated from the entire file will still be dissimilar from the feature vector of the actual benign executable. Hence, the crafted attack will not match with the benign executable.

An attacker may also insert redundant codes between sections or interchange the order of the sections which may result in a completely different fingerprint. This could be potentially addressed by analyzing sections of the code rather than the entire code, and computing localized descriptors.

We conduct a small experiment to see how much redundant information needs to be added to destroy the similarity. The experimental setup is as follows. L random vectors of length 2^M in the range $[0, 255]$ (8 bits) are generated. This is similar to files on disk whose bytes are random. We refer to these as "random noise patterns" and are indexed as $\{(F_i)\}_{i=1}^{L}$. From these, N files are generated as follows. K random "bytes" are inserted in α random positions for every F_i. Hence, N "variants" are generated from every F_i. We loosely model this setup as L classes with each class containing $N + 1$ samples (original pattern F_i + N variants). The parameters we choose are $L = 10$,$M = 12$,$N = 25$, $K = [1, 2, 3, 4, 5]$ and $\alpha = [2, 5, 10, 20, 50, 100, 200, 500]$. Fig. 4.29 shows

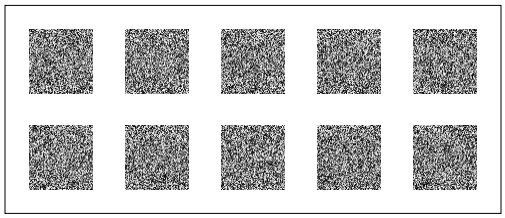


Figure 4.29: Visualization of random noise patterns ($\{(F_i)\}_{i=1}^{10}$)

the visualization of random noise patterns.

On this data, we compute GIST features and perform 10-fold cross-validation using Nearest Neighbors Classifier. The results are shown in Fig. 4.30. The accuracy is high when the number of alterations $\alpha = 50$. When $K = 1, \alpha = 50$ the accuracy is almost 100% and it drops as K increases. The accuracy is around 70% at $K = 5, \alpha = 250$ and this corresponds to adding approximately 6% of redundant data (since length of F_i = 4096). $K = 1, \alpha = 500$ and $K = 5, \alpha = 100$ correspond to adding around 12% of redundant data and the accuracies are around 60% and 40% respectively. This shows that adding fewer bytes in more positions preserves the similarity. At $\alpha = 500$ and $K = 2, 3, 4, 5$ (adding redundant data from 24% - 61%), the accuracy severely drops down. An adversary will have to insert this much of redundant data to destroy the similarity.

4.4.2 Strong Encryption

If some strong cryptographic methods, such as AES, are used by packers, it will be hard to find statistical similarity among such encrypted samples. If the block chaining is enabled, this problem becomes almost impossible. Our approach is unable to identify similarity in such cases. A possible counter measure to this would be to unpack/decrypt

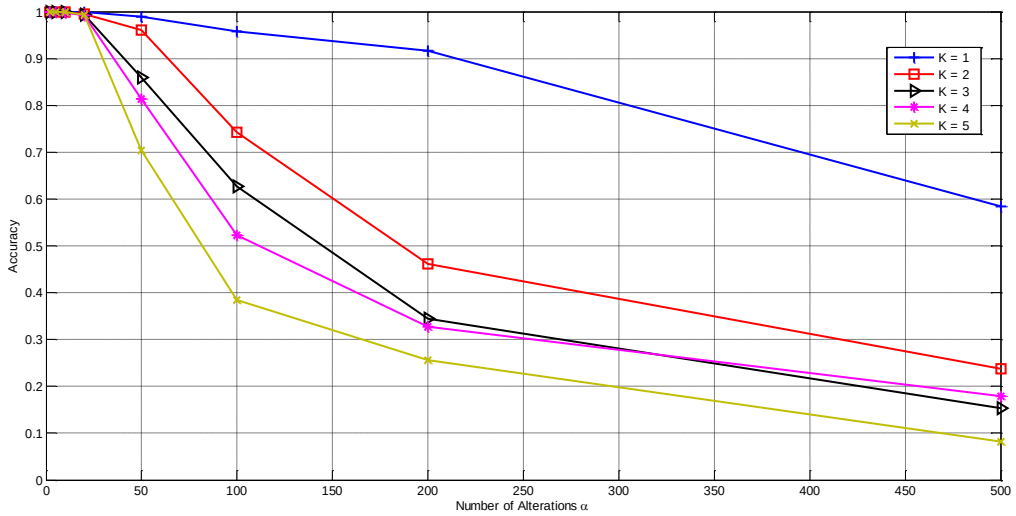


Figure 4.30: Evaluation of GIST features on random noise patterns

the malware in a virutal environment and then compute the image based feature. However, current packers use simple encryption and do not go for strong encryption. The use of strong encryption itself can be considered suspicious, which malware writers would want to avoid.

4.4.3 Patching

Malware can infect a benign executable by patching and embedding malicious code into it. If the embedded content is very small relative to the actual benign file-size, then the infected file is likely to be considered similar to the benign file. When we manually analyzed the false negatives in malware detection, we found that the majority of them were a case of patched system binary. For example, we found instances of TDSS rootkit that embeds its code into a small existing *.rsrc* section of Windows driver files, such as netbt.sys. We also found false positive cases of benign input files, which were

not present in the benign dataset, but its infected version was present in the malware dataset. These problems are also generic to file-similarity based detection techniques. One countermeasure can be flagging the input file as suspicious, if it is very similar to a system file, but not exactly the same file. However, if the training dataset contains a version of the infected file, or the benign file, detection is more likely to provide true-positive results.

4.5 Summary

In this chapter we described two systems that uses image similarity descriptors on malware binaries: SigMal for malware detection and SARVAM for malware retrieval. SigMal can operate on both packed and unpacked samples, avoiding the resource intensive unpacking process. Our experiments showed that SigMal outperforms all existing static malware detection methods in terms of precision. Large-scale experiments on 1.2 Million recent samples, both packed and unpacked, observed over three months demonstrated that our method can detect 50% of the recent incoming samples with above 99% precision. SARVAM, a system for content-based Search And RetrieVAl of Malware finds similar malware based on image similarity. On a database of more than 4 million malware, the average query time for SARVAM to find a match is 3 seconds. Our experiments showed that out of 212,000 samples that were uploaded, nearly 60% of the samples were variants of existing samples. Both our systems showed that using image similarity descriptors on malware is feasible for practical and large scale malware analysis.

Chapter 5

Sparsity based Malware Analysis

5.1 Introduction

In Chapter 3 and Chapter 4, we explored the possibility of representing malware binaries as digital images and using image similarity descriptors to analyze malware. In this chapter we generalize the 2D image based malware analysis approach to a 1D signal based approach by expanding malware as a sparse linear combination of other malware samples. We explore Sparse Representation based Classification (SRC) methods to classify malware variants into families and present *SATTVA: SpArsiTy inspired classificaTion of malware VAriants*. SRC methods have been previously applied to problems where samples belonging to a class have small variations in them, for example, face recognition [111], iris recognition [82], background subtraction [37], and tracking [70].

Exploiting the fact that most malware variants have small differences in their structure, we model a new/unknown malware sample as a sparse linear combination of other

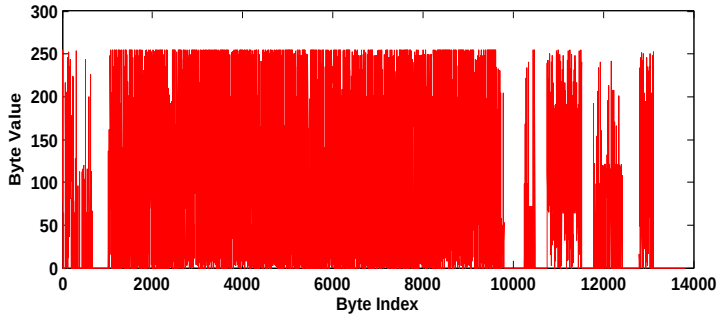
malware in the training set. The class with the least residual error is assigned to the unknown malware. Fig. 5.1 shows one such example of two malware variants of the recently exposed Regin malware [16], which has been described as one of the most sophisticated malware discovered in recent times, and termed on par with Stuxnet, Flame and other advanced malware. In Fig. 5.1, variants are represented as byte plots where every byte is represented as a number. Reports further observe that variants of Regin malware were used for diverse tasks such as cyber-espionage and secret surveillance against countries, companies and individuals. Although this example shows a case of simple mutation, this phenomenon is also true for variants created using executable packers, which are more common nowadays.

The rest of the chapter is organized as follows. Sec. 5.2 details the formulation of the sparse representation based classification framework. Sec. 5.3 details the experiments on various datasets. The limitations and summary are discussed in Sec. 5.4.

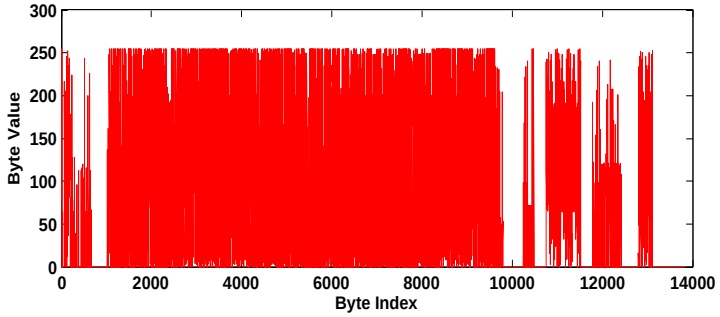
5.2 Malware Classification based on Sparse Representations

5.2.1 Approach

Given a dataset of N labeled malware belonging to L different malware families with P malware per family, the task is to identify the family of an unknown malware $\mathbf{u}$. Similar to [72], we represent a malware as a numerical vector $\mathbf{x}$ of range $[0, 255]$, where every entry of $\mathbf{x}$ is a byte value of the malware. However unlike [72], we do not convert this vector to an image matrix. Since each malware sample can have a different



(a) Variant 1



(b) Variant 2

Figure 5.1: Byte plots of the recently exposed Regin malware variants [16]. Variant (b) is created by making small change to Variant (a). They differ only in 7 bytes out of 13,284 bytes (0.0527 %).

code-length, we normalize all vectors to a maximum length (M) by zero-padding.

The entire dataset can now be represented as an $M \times N$ matrix $\mathbf{A}$, where every column represents a malware. Further, for every family k ($k = 1, 2, ..., L$), we define an $M \times P$ matrix $\mathbf{A}_k = [\mathbf{x}_{k1}, \mathbf{x}_{k2}, ...\mathbf{x}_{kP}]$ where $\mathbf{x}_{k\{.\}}$ represents a malware sample belonging to family k. Now, $\mathbf{A}$ can be expressed as a concatenation of block-matrices $\mathbf{A}_k$:

$$\mathbf{A} = [\mathbf{A}_1 \mathbf{A}_2 .. \mathbf{A}_L] \in \mathbb{R}^{M \times N} \tag{5.1}$$

Let $\mathbf{u} \in \mathbb{R}^M$ be an unknown malware whose family is to be determined, with

the assumption that $\mathbf{u}$ belongs to one of the families in the dataset. Since variants in a family have small differences, they will all be in the same linear span [1]. Then, following [111], we represent $\mathbf{u}$ as a sparse linear combination of the training samples as:

$$\mathbf{u} = \sum_{i=1}^{L} \sum_{j=1}^{P} \alpha_{ij} \mathbf{x}_{ij} = \mathbf{A}\alpha \tag{5.2}$$

where $\alpha = [\alpha_{1,1}, ..., \alpha_{L,P}]^T$ represents the $N \times 1$ sparse coefficient vector ($N = LP$). α will have non-zero values only for samples that are from the same family as $\mathbf{u}$. The sparsest solution to (5.2) can be obtained using Basis Pursuit [82] by solving the following l_1-norm minimization problem:

$$\hat{\alpha} = \arg\min_{\alpha' \in \mathbb{R}^N} \|\alpha'\|_1 \text{ subject to } \mathbf{u} = \mathbf{A}\alpha' \tag{5.3}$$

where $\|.\|_1$ is the l_1 norm.

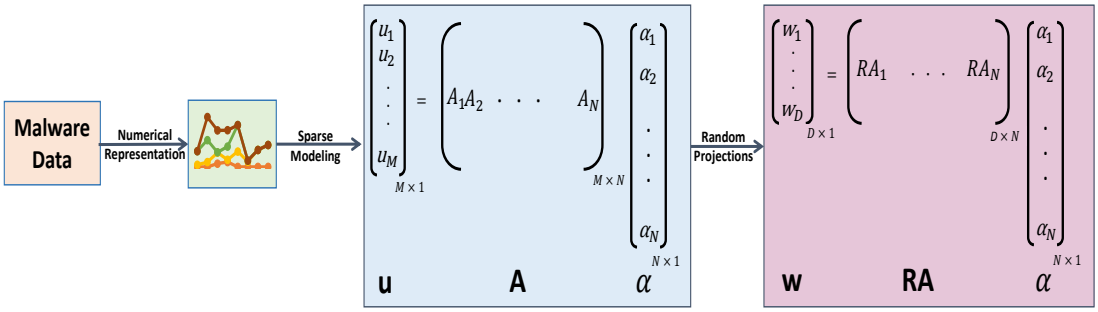
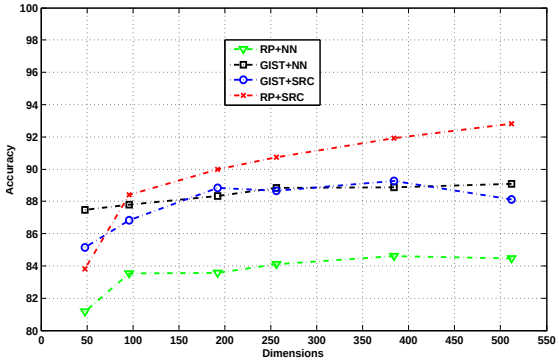


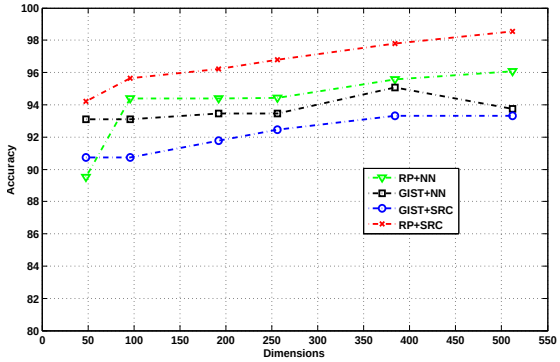
Figure 5.2: Overall Approach: Malware samples are represented as numerical vectors, projected to lower dimensions and then modeled using the Sparse Representation based Classification (SRC) framework

Estimating the family of $\mathbf{u}$ is done by computing residuals for every family in the training set and then selecting the family that has minimum residue. Let $\mathbf{\Pi}_k$ be the

[1] Linear span means that any linear combination of a vector will be in the same subspace



(a) Malimg Dataset



(b) Malheur Dataset

Figure 5.3: Experimental Results on (a) Malimg Dataset and (b) Malheur Dataset with features using Random Projections (RP) and GIST, and classification algorithm using Sparse Representation based Classification (SRC) and Nearest Neighbor (NN).

characteristic function that selects the coefficients from $\hat{\alpha}$ that are only associated with family k. Then the residual function r_k can be expressed as:

$$r_k(\mathbf{u}) = \|\mathbf{u} - \mathbf{A}\boldsymbol{\Pi}_k(\hat{\alpha})\|_2 \tag{5.4}$$

$$c = \arg\min_k \; r_k(\mathbf{u}) \tag{5.5}$$

where c is the index of the estimated family associated with $\mathbf{u}$.

5.2.2 Random Projections

When a malware binary is represented as a numerical vector by considering every byte, the dimensions of that vector can be very high. For example, a 1 MB malware has around 1 Million bytes and this could make the calculations computationally expensive. Hence, we project the vectors to lower dimensions using Random Projections (RP). This also removes dependency on any particular feature extraction method. Previous works have demonstrated that SRC is effective in lower-dimensional random projections as well, see [42, 111, 82]. Let $\mathbf{R} \in \mathbb{R}^{D \times M}$ be the matrix that projects $\mathbf{u}$ from signal space M to $\mathbf{w}$ of lower dimensional space D ($D << M$):

$$\mathbf{w} = \mathbf{R}\mathbf{u} = \mathbf{R}\mathbf{A}\alpha \tag{5.6}$$

The entries of $\mathbf{R}$ are drawn from a zero mean normal distribution. The above system of equations is underdetermined and sparse solutions can be obtained by reduced l_1-norm minimization:

$$\hat{\alpha} = \underset{\alpha' \in \mathbb{R}^N}{\arg\min} \|\alpha'\|_1 \text{ subject to } \mathbf{w} = \mathbf{R}\mathbf{A}\alpha' \tag{5.7}$$

The overall approach is shown in Fig.5.2.

5.2.3 Modeling Variants

When a new variant is created from existing an malware by making small changes, both variants share some common parts. The new variant is modelled as:

$$\mathbf{u}' = \mathbf{u} + \mathbf{e}_\mathbf{u} = \mathbf{A}\alpha + \mathbf{e}_\mathbf{u} \tag{5.8}$$

where $\mathbf{u}'$ is the corrupted vector representing the new variant and $\mathbf{e_u}$ is the error vector. This can be reduced to matrix form using block matrices:

$$\mathbf{u}' = [\mathbf{A}, \mathbf{I}_M] \begin{bmatrix} \alpha \\ \\ \mathbf{e_u} \end{bmatrix} = \mathbf{B_u}\ \mathbf{s_u} \tag{5.9}$$

where $\mathbf{B_u} = [\mathbf{A}, \mathbf{I}_M]$ is a $M \times (N + M)$ matrix and $\mathbf{I}_M$ is an $M \times M$ Identity matrix and $\mathbf{s_u} = [\alpha, \mathbf{e_u}]^T$. This ensures that the system of equations (5.9) is always under-determined and sparse solutions can be obtained. In lower dimensions, this reduces to:

$$\begin{aligned} \hat{\alpha} = \underset{\alpha' \in \mathbb{R}^N}{\arg\min} \|\alpha'\|_1 \text{ subject to } \mathbf{w}' = \mathbf{B_w}\ \mathbf{s_w} \\ r_k(\mathbf{w}') = \|\mathbf{w}' - \mathbf{B_w}\ \mathbf{s_w} \mathbf{\Pi}_k(\hat{\alpha})\|_2 \\ c = \underset{k}{\arg\min}\ r_k(\mathbf{w}') \end{aligned} \tag{5.10}$$

where $\mathbf{w}' = \mathbf{w} + \mathbf{e_w}$, $\mathbf{B_w} = [\mathbf{R}\mathbf{A}\alpha, \mathbf{I}_D]$ is a $D \times (N + D)$ matrix, $\mathbf{I}_D$ is a $D \times D$ Identity matrix and $\mathbf{s_w} = [\alpha, \mathbf{e_w}]^T$ We will use (5.10) to identify the malware family of an unknown test sample.

5.3 Experiments

We test our technique on two public malware datasets: Malimg Dataset [72] and Malheur Dataset [86]. On both datasets, we select equal number of samples to reduce any bias towards a particular family [65]. The data is converted to numerical form and represented as a matrix as defined in (5.1) and then projected to lower dimensions using Random Projections (RP). For comparison, we use GIST features [77], which

have been previously applied for malware classification [72]. We use the SRC framework (5.10) to identify the malware family of a test sample and compare with Nearest Neighbors (NN) classification that was previously used in [72]. We vary the dimensions from $\{48, 96, 192, 256, 384, 512\}$, which are consistent for both RP and GIST. In our experiments, we chose 80% of a dataset for training and 20% for testing.

5.3.1 Classification

Results on Malimg Dataset: The Malimg dataset contains 25 malware families with 9,342 samples, which we obtained from the authors of [72]. The dataset has a mixture of both packed and unpacked malware and the number of samples per family varies from 80 to 2,949. In our experiments, we select 80 samples per family (the minimum number present in all families). The size of the largest malware (M) was 840,960 bytes and all samples were zero padded to this size. The results are shown in Fig. 5.3a. First, we see that the classification accuracy increases as the dimensionality increased from 48 to 512. Beyond 512, there was no significant change in accuracy for both GIST and RP. The best accuracy of **92.83%** was obtained for RP with SRC as the classifier. At the same dimension, the lowest accuracy was for RP with NN as the classifier (84.45%). The accuracies for GIST for both classifiers were almost the same, in the middle range (88-89%).

Results on Malheur Dataset: The Malheur dataset consists of 3,131 malware binaries from 24 malware families, which we obtained from the authors of [86]. The malware binaries were labeled such that a majority amongst six different antivirus products shared similar labels. The number of samples per family varied between 20 and 300. We chose 20 samples from all families in our experiments. For this dataset, the value

of M was 3,364,864. The classification results are shown in Fig. 5.3b. Here too, the best accuracy of **98.55%** was obtained for RP at 512 dimensions with SRC as classifier. However, unlike the Malimg dataset, RP with NN as classifier also had a high accuracy of 96.06%. This shows that the random projections of the variants in Malheur dataset are closely packed in lower dimensions. On the other hand, the accuracies for GIST features were around 93% for both classifiers.

5.3.2 Comparison with Other Features

We have already looked into related works using statistical features in Chapter 2 (Sec. 2.1.3). Common statistical features are based on n-grams [58, 52, 53], n-perms [54, 62], hashes [60, 107] and image similarity [72, 76, 75, 55]. In contrast to these methods, we compute random projections on malware represented as digital signals. This results in compact features for malware classification. Although random projections have been previously used in [50, 40], these methods require dynamic analysis which is time consuming.

We compare our proposed approach with other relevant malware similarity features: *ssdeep* [60], GIST [72] and n-grams [58]. For n-grams, we chose $n = 2$ and computed a 2^{16} dimensional feature vector. The results are shown in Tab. 5.1. For both datasets, our proposed approach outperformed *ssdeep*, GIST and n-grams based features.

Table 5.1: Comparison with Other Features

Dataset	*ssdeep*	GIST	n-grams	RP
Malimg Dataset	67.63	89.08	91.75	**92.83**
Malheur Dataset	81.6	94.21	94.26	**98.55**

5.3.3 Rejecting Outliers

In order to reject test samples that do not belong to any family in a dataset, the Sparsity Coefficient Index (SCI) of a coefficient vector $\alpha \in \mathbb{R}^N$ is defined as:

$$SCI(\alpha) = \frac{\frac{L.max\|\mathbf{\Pi}_i(\alpha)\|_1}{\|\alpha\|_1} - 1}{L - 1} \tag{5.11}$$

The value of SCI varies between 0 and 1, 1 being the test sample can be represented as a linear combination of one family and 0 being the test sample is spread across all the families. It is common to have a threshold $\tau \in (0, 1)$ and reject outliers that are below τ.

For the Malimg Dataset, we vary τ for a fixed dimension ($D = 512$) as shown in Fig. 5.4a. At $\tau = 0.1$, the accuracy is 92.5% with no samples rejected. Accuracy of 100% is achieved when $\tau = 0.5$, at which 25% of the samples are rejected from the dataset. Similarly, for the Malheur dataset, we computed the accuracies and the percentage of samples dropped while varying τ. In Fig. 5.4b, we see that accuracy of 100% is reached when $\tau = 0.6$, but with only 5% of samples rejected.

5.3.4 Approximate l_1-norm

So far, we have used Basis Pursuit (BP) [38] for l_1-norm minimization and to recover the sparse coefficients. However, BP is computationally expensive and is not suitable for large scale data. Here, we compare the computation time and accuracy obtained using BP with an approximate l_1-norm minimization method, Orthogonal Matching Pursuit (OMP) [104]. OMP is a greedy algorithm that works by iteratively selecting a subset of columns from the training data matrix that are almost orthogonal. We repeat the experiments on both datasets using OMP and report the time taken to identify the

families of all samples in the test set. The results are shown in Tab. 5.2. We see that for both datasets, the computation time decreased by a factor of 18 (Malimg) and 30 (Malheur) respectively, at the cost of slight decrease in classification accuracy. This makes OMP suitable for large scale malware classification.

Table 5.2: Basis Pursuit (BP) vs Orthogonal Matching Pursuit (OMP)

Dataset	BP Accuracy	OMP Accuracy	BP Comp. Time (s)	OMP Comp. Time (s)
Malimg Dataset	92.83	89.25	420	**24**
Malheur Dataset	98.55	97.39	180	**6**

5.3.5 Analysis on Large Scale Data

We evaluated our technique on two diverse large scale datasets. On both datasets, we randomly selected 20% of the data for testing and used Orthogonal Matching Pursuit to find the sparse coefficients. The results on both datasets show that our technique is applicable in large scale scenarios.

Results on Offensive Computing Dataset: We downloaded more than 1.4 Million malware from the Open Malware sharing platform [12] (formerly known as Offensive Computing). The samples were fed to different Antivirus software for labeling and the software that had minimum number of unknown labels was selected. This resulted in 2,124 malware families and we randomly selected 20 samples from each family to obtain a dataset of 42,480 samples (20 was the minimum number of samples present in some families). The size of the largest malware was 9.3 MB. We repeated the experiments using OMP and obtained an average classification accuracy of **66.34%**. The overall testing time was approximately 4 hours on a standard desktop machine. This time can further be reduced by using parallelization techniques. Out of 2,124 fami-

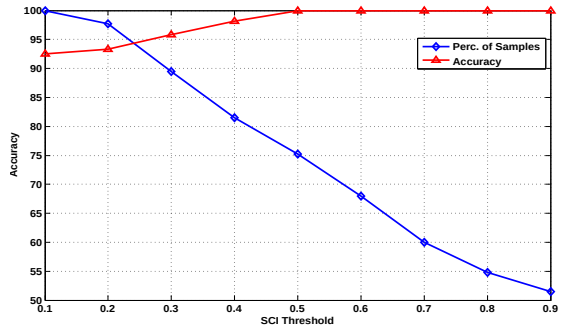
lies, **927** families had an accuracy of **100%**. The average SCI value for these families was 0.97, with most values being 1. This shows that SCI can be used as a confidence measure during testing. At an SCI threshold of 0.6, 24.78% of the test samples were rejected and the classification accuracy was **77.08%**.

Results on Anubis Dataset: Next, we evaluated our technique on another large scale dataset that we obtained from the authors of Anubis [26]. The Anubis dataset had 36,784 samples divided into 209 clusters, with 176 samples per cluster. The clusters were labeled according to the behavioral pattern of a malware upon dynamic analysis [26]. This dataset is different from the Offensive Computing dataset in two aspects. First, the number of samples in a family/cluster is higher. Second, the labeling of clusters is based on dynamic analysis. This means there is a possibility that two samples that have very different structure but similar behavior can be assigned the same cluster, and our technique will not work on such samples. For this dataset, the maximum size of the malware was 8.1 MB. On repeating the experiment, we obtained an average classification accuracy of **57.36%**. This is much lower than the accuracy obtained for the Offensive Computing dataset, which had more number of classes (by a factor of 10). This perhaps shows that our method is better applicable to malware datasets that have finer labels. The overall testing time was approximately 3 hours on a standard desktop. For this dataset, **27** clusters had an accuracy of **100%** and 50 clusters had an accuracy of more than 90%. On setting the SCI threshold to 0.6, 34.64% of the test samples were rejected and we obtained an accuracy of **77.12%**.

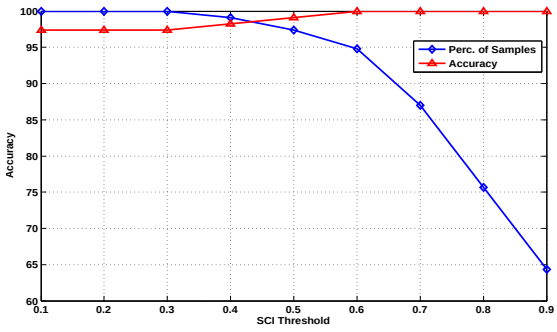
5.4 Summary

Our approach works well mainly on malware variants that have similar structure. However, we observe that most variants are those that are structurally similar (for example, Regin variants in Fig. 5.1). This is also evident from our large scale experiments.

In this chapter we proposed a novel method to identify families of malware variants using a combination of Sparse Representation based Classification (SRC) and Random Projections (RP). Experiments on two standard malware datasets, as well as large scale data showed promising results. We believe that our approach, that is based on representing malware binaries as numerical signals, will open the scope of malware analysis to broader fields.



(a) Malimg Dataset



(b) Malheur Dataset

Figure 5.4: Rejecting outliers based on Sparsity Coefficient Index (SCI). Higher the value of SCI, higher the classification accuracy. Both datasets achieve 100% accuracies at an SCI value of 0.6. For the Mallheur dataset, only 5% of samples are rejected to achieve this accuracy. However, for the Malimg dataset, nearly 32% of samples are rejected for the same.

Chapter 6

Extensions to Data Type Classification

6.1 Introduction

In this chapter we extend our previous approaches to the problem of *data type classification*: to determine the type of data from a block of data. This is an important problem in data forensics tasks such as data recovery from corrupted file systems, understanding process memory dumps, reverse engineering and others. We adopt a two stage meta learning framework to identify data types (Sec. 6.3). In the first stage, statistical and content based features are computed. Using a classifier, class probability vectors are computed for every feature. In the second stage, the class probability features are *stacked* to form a new feature vector which is fed to another classifier for the final classification. These types of *stacking* and *blending* of feature predictions and models are used in practical machine learning problems and competitions [59, 103, 13]. Further, we use this approach to identify families of malware variants after they are compressed. In this chapter we present *MAYAM: Meta leArning based data tYpe And*

compressed Malware classification. There has been an increase in attacks by using compressed malware [14, 8, 7, 15]. However, even after malware variants are compressed, there are similarities in their structure. We demonstrate the efficacy of our approach with extensive experiments.

The rest of the chapter is organized as follows. The related works in data type classification and malware classification are briefed in Sec. 6.2. Sec. 6.3 details the meta learning based classification framework. Sec. 6.5 details the experiments on various datasets. The summary is presented in Sec. 6.6.

6.2 Related Work on Data Type Classification

File type classification is the determination of the type of a file by examining the content of blocks of data in the absence of file type specific information such as headers and magic numbers. Examples of file types include doc, ppt, pdf, jpeg and others. File fragment classification or data type classification is the determination of the ”data type” of a particular block of data in the absence of any information such as headers and magic numbers. Examples of data types include txt, bmp,csv, gzip, lzma and others. A file can have many blocks of data that are of different data types. Sometimes the file type and data type can be the same in the case of files such as bmp and csv. There are several previous works on file and data type classification [69, 97, 35, 45, 23, 47, 88, 39, 22, 48, 46, 28, 89, 83]. We will briefly review some of the relevant works on content based file type and data type classification below.

In one of the first works in file type classification, Mcdaniel and Heydari [69] used byte frequency analysis, byte frequency cross correlation to determine the file type.

They also analyzed the header and trailer of a file and showed that fingerprints based on the header and trailer had better performance than the fingerprints based on byte frequency and their correlation. However, the header and trailer may not be always available and an analysis that does not take header and trailer into account is more reliable. Stolofo et. al [97] developed fingerprints for different file types called Fileprints using 1-grams. S. Garfinkel et. al [47] used features based on 1-grams and 2-grams to classify file types. From a corpus of more than 1 Million files, they selected random subsets for their experiments. On 24 different types, they were able to obtain an average classification accuracy of 49%. In [39], Conti et. al used statistical quantities such as mean, entropy, chi squared distance and Hamming weight to obtain a 4 dimensional vector. Using a k-Nearest Neighbor classifier, they classified 14 file fragments and obtained an average accuracy of 64.2%. N. Beebe et. al [28] proposed a technique called Scaeandan which used concatenated n-grams (1-grams and 2-grams) and use SVM for classifying file types. A total of 40+ file types were used and they obtained an average classification accuracy of 73.4%. A careful analysis of their results showed that certain file fragment types obtained very high accuracies and were comparable with other previously proposed methods. These types include csv, ps, gif, sql, html, css and plain text. However, there were also a number of fragment types which obtained very low accuracies such as ppt, xlsx, docx, pps, pptx, wmv, and pdf.

We had previously reviewed past works in malware classification in Chapter 2(Sec. 2.2). The common statistical and content based features are based on n-grams [58, 53], hashes [60, 107], image similarity [72] or signal similarity [74]. There are no previous works in classification of compressed malware.

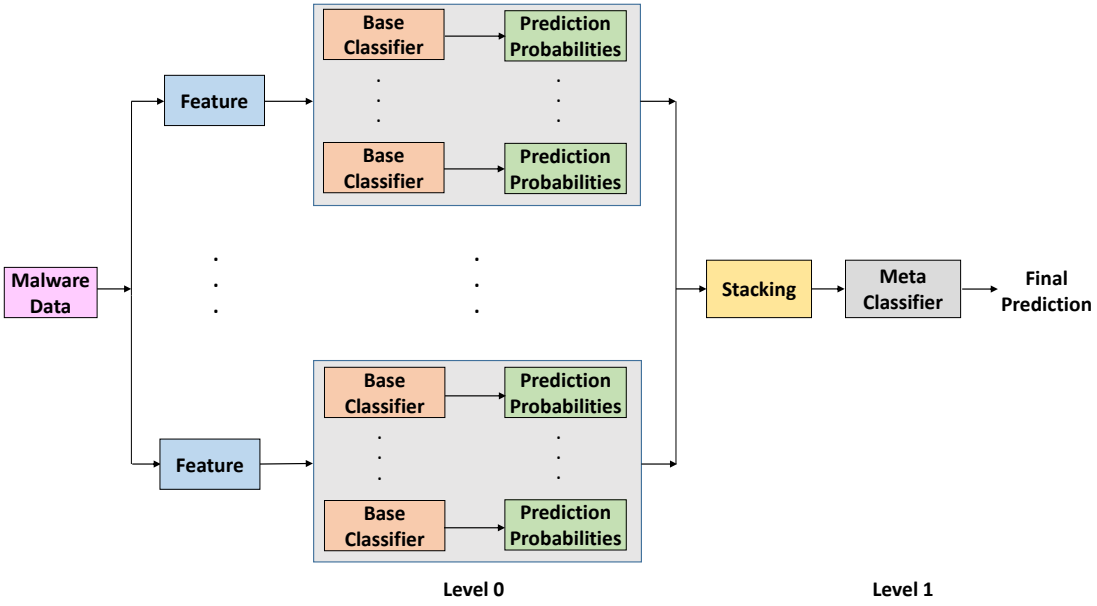


Figure 6.1: Block Schematic of Stacked Generalization

6.3 Classification based on Meta Learning

Meta Learning is a two staged classification framework. In the first stage, a dataset is divided into disjoint sets or folds. At every iteration, one fold is treated as the testing set and other folds are treated as the training set. Features are extracted and passed through one or many classifiers. These are called *base* (*level-0*) features and classifiers. The output of the first stage is prediction labels or probabilities. In the second stage, a *meta* (*level-1*) learner uses the output from the first stage to boost the prediction accuracy. For a good survey on meta learning, readers are referred to [29].

Stacked Generalization [110, 109, 31, 30] is a meta learning based technique that stacks the prediction outputs from the base level to form a new feature vector, which is then fed to a meta learner. This approach has shown promising results in many diverse machine learning applications [99, 98, 100, 94, 71, 112, 13]. A block schematic of

stacked generalization is shown in Fig. 6.1.

The steps involved in our approach are:

1. Divide the data into k folds, with $k-1$ folds being the training and k-th fold being the testing set.
2. Compute the features F_i on all the samples.
3. Using a base classifier, obtain the prediction probabilities for the features F_i of the testing set.
4. Repeat the above steps by varying the testing set and training set for different fold
5. Stack the prediction probabilites from different F_i to form the meta feature
6. Compute the final prediction by using meta features on meta classifier.

6.4 Features

We extract two types of features- statistical features and content based features. Statistical features are known to work well on classifying data types while content based features shave shown promising results on classifying malware families.

Statistical Features

2-grams features: Let b_g denote the bigram count of every consecutive byte pair. The total number of possible combinations is $2^16 = 65,436$. Since the dimensionality is high, we project b_g to lower dimensions D using Random Projections. Let R be a

pseudo-random matrix of dimensions $Dx65,436$. We define $F_1 = R.b_g$, where F_2 is now a D-dimensional feature vector. In our experiments, we choose $D = 512$. Beyond this value, there was no significant difference in performance. Both these statistical features are normalized.

1-grams features: Let F_2 denote the histogram count or 1-grams of every byte in a binary or file fragment. The dimensionality of F_2 is 256.

Content based Features

For content based features, we use GIST descriptors [77], which have been previously applied for malware classification [72]. The binary content is first represented as a gray-scale image and then re-sized to a square image. This image is then passed through various sub-bands of different scale and orientation from which filtered images are obtained. The filtered images are then divided into sub-blocks and the average value of each sub-block is computed. The final descriptor is a concatenation of all the average values. Let F_3 denote the GIST feature vector whose dimension is 320.

6.5 Experiments

We use the stacking framework (Sec. 6.3) to classify data types and compressed malware. In the first stage, we divide the data into 10-folds and use Random Forests [32] as the base level classifier. In the second stage, we stack the prediction probabilities from the first stage to form a new feature vector. This is then fed to a Linear Support Vector Machine (SVM) [34] with stochastic gradient descent optimization for the final classification. All experiments are performed using 10-fold cross validation.

Table 6.1: Classification of USMA File Fragment Dataset

Data Type	**Conti et al. [39]**	F_{meta}
bitmap	82.5	**97.7**
compress_bzip2_text	30.6	**81.1**
compress_compress_text	58.8	**79.9**
crypt_aes256_text	38.6	**45**
deflate_png	42.4	**59.6**
exe_linux_elf_text	82.3	**98.2**
exe_windows_pe_text	72.1	**96.4**
jpg	44.1	**81.6**
lzw_gif	66.1	**80.1**
music	45.5	**88.9**
random_pseudo	37.5	33
text	98.7	97.7
zip_base64	100	100
zip_uuencode	100	100

6.5.1 Classification of Data Types

The USMA File Fragment Dataset consists of 14 data types which we obtained from the authors of [39]. There are 14,000 file fragments in total, with 1,000 fragments for each data type. The classification results are shown in Tab. 6.1. For comparison, we report the results from [39]. Our proposed method outperformed for 10 data types and equaled Conti et al.'s method for 2 data types while the difference in accuracy is small for other 2 data types.

6.5.2 Classification of Compressed Malware

In this experiment, we compress the Malheur dataset [86] and Malimg dataset [72] using the following compression methods: *gzip*, *xz* and *bzip2*. After compression, we also use the following archival techniques: *tgz*, *txz*, *tbz2*, *zip* and *zip* with password

Table 6.2: Classification of Malheur Dataset after compression

Compression Type	F_1	F_2	F_3	F_{meta}
nil	98.18	97.95	98.46	**99.23**
gzip	77.77	87.16	92.39	**96.8**
xz	67.23	67.42	89.14	**91.12**
bzip2	50.91	70.07	86.14	**93.23**
zip	76.14	87.35	92.81	**96.96**
zip-pwd	46.12	44.62	79.14	**82.05**
tgz	71.96	85.09	91.33	**96.56**
txz	42.76	43.78	80.83	**82.49**
tbzip2	85.62	70.36	48.83	**92.46**

(*zip-pwd*).

Results on Malheur Dataset: The Malheur reference dataset consists of 3,131 malware binaries from 24 malware families, which we obtained from the authors of [86]. The malware binaries were labeled such that a majority amongst 6 different antivirus products shared similar labels. We compressed this dataset using 8 different compression and archival techniques. The classification results are shown in Tab. 6.2. Our proposed method outperforms the classification accuracies of individual features in all cases. Other than (*zip-pwd* and *txz*, the accuracies for other 6 compression techniques are very high.

Results on Malimg Dataset: The Malimg dataset contains 25 malware families with 9,342 samples [72]. The dataset has a mixture of both packed and unpacked malware. This dataset was using 8 different compression and archival techniques, and the classification results are shown in Tab. 6.3. Our proposed method outperforms the classification accuracies of individual features in all cases. Similar to Malheur dataset, apart from (*zip-pwd* and *txz*, the accuracies for other 6 compression techniques are very high.

Table 6.3: Classification of Malimg Dataset after compression

Compression Type	F_1	F_2	F_3	F_{meta}
nil	84.37	97.00	98.24	**99.59**
gzip	43.44	73.79	87.12	**95.01**
xz	70.2	73.41	86.72	**90.65**
bzip2	57.88	68.24	76.40	**91.06**
zip	71.39	74.95	86.77	**91.41**
zip-pwd	50.89	51.04	67.68	**74.79**
tgz	59.37	70.06	83.92	**91.57**
txz	47.58	51.59	69.58	**75.03**
tbzip2	57.47	68.20	75.86	**90.95**

Table 6.4: Classification of Malimg Dataset after compression with extra Benign class

Compression Type	F_1	F_2	F_3	F_{meta}
gzip	69.24	74.38	86.22	**92.53**
xz	72.91	75.36	87.54	**91.35**
bzip2	61.49	70.10	76.78	**91.18**
zip	72.88	76.89	87.51	**92.25**
zip-pwd	56.30	56.06	69.58	**75.04**
tgz	62.33	70.81	82.68	**90.88**
txz	54.94	57.79	70.76	**74.75**
tbzip2	61.33	70.26	76.65	**90.89**

Addition of Benign Class

In this experiment we added an extra benign set of 2,000 benign software (Windows System executables) to the Malimg dataset to see if our method can still distinguish between malware and benign. The results are shown in Tab. 6.4 and the addition of benign class did not affect our approach.

6.6 Summary

In this chapter we proposed a novel approach to classify data types and compressed malware using a two stage meta learning approach. Experiments on various datasets showed promising results. Some future research directions are adding more features and classifiers in the base level to obtain a richer feature description for the meta level.

Chapter 7

Conclusions and Future Work

In this thesis we explored orthogonal yet complementary methods to analyze malware motivated by Signal and Image Processing. Malware samples are represented as images or signals. Image and signal based features are extracted to characterize malware. With extensive experiments, we demonstrated the efficacy of our methods on malware classification, detection and retrieval. Finally, we extend our approaches beyond malware, to the field of data forensics and data type classification. We believe that our techniques will open the scope of signal and image based methods to broader fields in computer security.

In Chapter 3 we treat malware as 2D grayscale images and use image similarity descriptors to characterize a malware. This was based on the observation that malware variants of the same malware family were similar in structure and visual appearance. Since malware authors use similar techniques in creating variants of malware belonging to different operating systems, our method does not need to be re-developed for a particular platform. Our approach is fast since there is no need for disassembly or

execution of the malware. Hence it is faster than most static analysis and dynamic analysis based techniques. Since the structure of packed malware variants do not change after packing, our method is able to find similarity among packed malware variants where static analysis approaches like control flow graph analysis fail. Malware authors use similar obfuscation techniques when creating malware variants of non-Windows based Operating Systems Linux, Android and OS X. Hence our method need not be re-developed for a particular Operating System while traditional static and dyanamic analysis based techniques need to be re-developed.

In Chapter 4 we demonstrate the feasibility of image similarity to malware detection and retrieval. We presented SigMal, a framework for fast signal processing based malware triage and showed how our approach outperformed other static and statistical features. We developed an online malware search and retrieval system, SARVAM, where one can upload a malware and retrieve similar malware from a database of over 7 million. Since 2012, we have received more than 440,000 samples of which nearly 60% were variants of malware in our database.

In Chapter 5 we treat malware as 1D signals. Since variants from the same malware family have small changes, we modeled a malware variant as a sparse linear combination of other variants from the same family. Further, we reduced the dimensions of the malware signals using Random Projections and showed that lower dimensional projections also preserved the similarity among variants. Our experimental results on various datasets showed that this generalized approach had better classification performance than the image based method.

In Chapter 6 we extended our previous approaches to data type classification and compressed malware similarity. Instead of using a single feature, we used many fea-

tures and stacked the prediction scores to form a meta level feature, which we use for classification. We showed how this two level meta learning based approach can be used for data type and compressed malware classification.

7.0.1 Future Directions

Extensions to other signal similarity metrics

While we represented malware as images and signals, a natural complement is to treat the malware binaries as audio-like one dimensional signals and leverage automated audio descriptors. Audio descriptors have shown great success in matching songs under conditions of limited length, low quality recording and high noise. Commercial applications such as Shazam (for identifying songs currently playing) and YouTube (to find uploaded songs subject to copyright) are well known examples. Similar to visual features, there are several audio features designed to extract key information from audio signals such as spectral flatness, MFCC (mel-frequency cepstrum coefficients), chroma features and more. Each capture a distinct feature of sound, and have varying applications such as exact match vs similar sounding, depending on respective strengths and weaknesses. As a preliminary experiment we tried chroma features on standard malware datasets , which yielded a classification accuracy of more than 90%. We believe that other audio features will show promising results.

Section based bag of malware signatures

Our main focus in this thesis has been computing malware signature on the whole executbale. However, a malware executable comprises many sections. In SigMal we saw that concatenation of features based on the whole executable and code sections outperformed the features computed on the full binary. Computing image similarity

descriptors and/or random projections on all the sections will have a richer description. These can then be represented as bag of malware signatures which can then be used to better characterize malware.

Exact location of malware

Using the error model in the sparse representation based malware classification framework, we can determine the exact positions in which the malware variant differs from another variant. This approach can also be used to find the exact source from which a malware variant evolves. Patched malware that attaches to benign software can be identified using this method.

Meta Learning of Static Features

Using statistical and content based features followed by meta learning, we saw how compressed malware could be classified. One could also use other static based features based on disassembly of code along with statistical and content based features for even better performance. In Chapter 6 all our experiments showed that meta level classification was always better than individual feature prediction. Stacking of feature predictions from several static and statistical features should outperform all the individual predictions.

Bibliography

[1] Anubis. http://anubis.iseclab.org/.

[2] contagio: OSX malware and exploit collection. http://contagiodump.blogspot.com/2013/11/osx-malware-and-exploit-collection-100.html.

[3] Cuckoo Sandbox. http://www.cuckoosandbox.org/.

[4] G DATA MOBILE MALWARE REPORT, THREAT REPORT: Q1/2015. https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q1_2015_US.pdf.

[5] GIST Code, http://people.csail.mit.edu/torralba/code/spatialenvelope. http://people.csail.mit.edu/torralba/code/spatialenvelope.

[6] Google's VirusTotal puts Linux malware under the spotlight. http://www.zdnet.com/article/googles-virustotal-puts-linux-malware-under-the-spotlight/.

[7] Malicious spam continues to serve zip archives of javascript files. https://isc.sans.edu/diary/Malicious+spam+continues+to+serve+zip+archives+of+javascript+files/19973.

[8] Malware uses zws compression for evasion tactic. http://blog.trendmicro.com/trendlabs-security-intelligence/malware-uses-zws-compression-for-evasion-tactic/.

[9] Malwr. http://malwr.com.

[10] Multi-dimensional Scaling, DR Toolbox. http://homepage.tudelft.nl/19j49/.

[11] Norman sandbox. http://sandbox.norman.no.

[12] Offensive Computing Dataset. http://offensivecomputing.net.

[13] Otto product classification winner's interview: 2nd place, alexander guschin. http://blog.kaggle.com/2015/06/09/otto-product-classification-winners-interview-2nd-place-alexander-guschin/.

[14] Phishing with a malicious .zip attachment. http://phishme.com/malware-analysis-zip-attachment/.

[15] Project camberdada - nsa. https://theintercept.com/document/2015/06/22/project-camberdada-nsa/.

[16] Regin: Top-tier espionage tool enables stealthy surveillance. http://www.symantec.com/connect/blogs/regin-top-tier-espionage-tool-enables-stealthy-surveillance.

[17] THE EVOLUTION OF OS X MALWARE. https://eugene.kaspersky.com/2014/09/29/the-evolution-of-os-x-malware/.

[18] VirusShare. http://www.virusshare.com.

[19] VirusTotal. http://www.virustotal.com.

[20] VX Heavens. http://vx.netlux.org.

[21] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41–42. IEEE, 2004.

[22] Irfan Ahmed, Kyung-Suk Lhee, Hyun-Jung Shin, and Man-Pyo Hong. Fast content-based file type identification. In *Advances in Digital Forensics VII*, pages 65–75. Springer, 2011.

[23] Mehdi Chehel Amirani, Mohsen Toorani, and A Beheshti. A new approach to content-based file type detection. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, pages 1103–1108. IEEE, 2008.

[24] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007, Proceedings*, volume 4637 of *Lecture Notes in Computer Science*, pages 178–197, 2007.

[25] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. Symp. Network and Distributed System Security (NDSS)*, 2009.

[26] U. Bayer, P.M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2009.

[27] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of NDSS*, 2009.

[28] Nicole L Beebe, Laurence A Maddox, Lishu Liu, and Minghe Sun. Sceadan: Using concatenated n-gram vectors for improved file and data type classification. *IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY*, 8(9):1519, 2013.

[29] Pavel Brazdil, Christophe Giraud Carrier, Carlos Soares, and Ricardo Vilalta. *Metalearning: Applications to data mining*. Springer Science & Business Media, 2008.

[30] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[31] Leo Breiman. Stacked regressions. *Machine learning*, 24(1):49–64, 1996.

[32] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[33] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 129–143. Springer, 2006.

[34] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Knowledge Discovery and Data Mining*, 2(4):121–167, 1998.

[35] William C Calhoun and Drue Coles. Predicting the types of file fragments. *Digital Investigation*, 5:S14–S20, 2008.

[36] E. Carrera and G. Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Virus Bulletin Conference*, pages 187–197, 2004.

[37] Volkan Cevher, Aswin Sankaranarayanan, Marco F Duarte, Dikpal Reddy, Richard G Baraniuk, and Rama Chellappa. Compressive sensing for background subtraction. In *Computer Vision–ECCV 2008*, pages 155–168. Springer, 2008.

[38] Scott Shaobing Chen, David L Donoho, and Michael A Saunders. Atomic decomposition by basis pursuit. *SIAM journal on scientific computing*, 20(1):33–61, 1998.

[39] Gregory Conti, Sergey Bratus, Anna Shubina, Benjamin Sangster, Roy Ragsdale, Matthew Supan, Andrew Lichtenberg, and Robert Perez-Alemany. Automated mapping of large binary objects using primitive fragment type classification. *digital investigation*, 7:S3–S12, 2010.

[40] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3422–3426. IEEE, 2013.

[41] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[42] David Donoho and Jared Tanner. Counting faces of randomly projected polytopes when the projection radically lowers dimension. *Journal of the American Mathematical Society*, 22(1):1–53, 2009.

[43] Matthijs Douze, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg, and Cordelia Schmid. Evaluation of gist descriptors for web-scale image search. In *Proceedings of the ACM International Conference on Image and Video Retrieval*, page 19. ACM, 2009.

[44] T. Ebringer, L. Sun, and S. Boztas. A fast randomness test that preserves local detail. In *Virus Bulletin*, 2008.

[45] Robert F Erbacher and John Mulholland. Identification and localization of data types within large-scale file systems. In *Systematic Approaches to Digital Forensic Engineering, 2007. SADFE 2007. Second International Workshop on*, pages 55–70. IEEE, 2007.

[46] Simran Fitzgerald, George Mathews, Colin Morris, and Oles Zhulyn. Using nlp techniques for file fragment classification. *Digital Investigation*, 9:S44–S49, 2012.

[47] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *digital investigation*, 6:S2–S11, 2009.

[48] Siddharth Gopal, Yiming Yang, Konstantin Salomatin, and Jaime Carbonell. Statistical learning for file-type identification. In *Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on*, volume 1, pages 68–73. IEEE, 2011.

[49] F. Guo, P. Ferrie, and T. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection (RAID)*, 2008.

[50] Jozsef Hegedus, Yoan Miche, Alexander Ilin, and Amaury Lendasse. Methodology for behavioral-based malware analysis and detection using random projections and k-nearest neighbors classifiers. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1016–1023. IEEE, 2011.

[51] X. Hu, T. Chiueh, and K.G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[52] G. Jacob, P.M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware sample. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*. Springer, 2012.

[53] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011.

[54] M.E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, 2005.

[55] Dhilung Kirat, Lakshmanan Nataraj, Giovanni Vigna, and B.S. Manjunath. Sigmal: A static signal processing based malware triage. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec 2013.

[56] Clemens Kolbitsch, Paolo Milano Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyang Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of Usenix Security*, 2009.

[57] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[58] J.Z. Kolter and M.A. Maloof. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 7:2721–2744, 2006.

[59] Yehuda Koren. The bellkor solution to the netflix grand prize. *Netflix prize documentation*, 81, 2009.

[60] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

[61] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[62] A. Lakhotia, A. Walenstein, C. Miles, and A. Singh. Vilo: a rapid learning nearest-neighbor classifier for malware triage. *Journal of Computer Virology and Hacking Techniques*, 9(3):109–123, 2013.

[63] Tony Lee and Jigar Mody. Behavioral classification. In *EICAR*, 2006.

[64] Peng Li, Limin Liu, Debin Gao, and Michael K. Reiter. On challenges in evaluating malware clustering. In *Proceedings of RAID*, 2010.

[65] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. On challenges in evaluating malware clustering. In *Recent Advances in Intrusion Detection*, pages 238–255. Springer, 2010.

[66] R. Lyda and J. Hamrock. Using entropy analysis to find Encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, Mar 2007.

[67] B. S. Manjunath and W.Y. Ma. Texture features for browsing and retrieval of image data. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI - Special issue on Digital Libraries)*, 18(8):837–42, Aug 1996.

[68] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of RAID*, 2008.

[69] M. McDaniel and M.H. Heydari. Content based file type detection algorithms. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10 pp.–, Jan 2003.

[70] Xue Mei and Haibin Ling. Robust visual tracking and vehicle classification via sparse representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(11):2259–2272, 2011.

[71] Eitan Menahem, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Improving malware detection by applying multi-inducer ensemble. *Computational Statistics & Data Analysis*, 53(4):1483–1494, 2009.

[72] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, pages 4:1–4:7, New York, NY, USA, 2011. ACM.

[73] Lakshmanan Nataraj, Grégoire Jacob, and BS Manjunath. Detecting packed executables based on raw binary data. Technical report.

[74] Lakshmanan Nataraj, S Karthikeyan, and BS Manjunath. Sattva: Sparsity inspired classification of malware variants. 2015.

[75] Lakshmanan Nataraj, Dhilung Kirat, BS Manjunath, and Giovanni Vigna. Sarvam: Search and retrieval of malware. In *Proceedings of the Annual Computer Security Conference (ACSAC) Worshop on Next Generation Malware Attacks and Defense (NGMAD)*, 2013.

[76] Lakshmanan Nataraj, Vinod Yegneswaran, Phillip Porras, and Jian Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM workshop on Security and Artificial Intelligence*, AISec '11, pages 21–30, New York, NY, USA, 2011. ACM.

[77] A. Olivia and A. Torralba. Modeling the shape of a scene: a holistic representation of the spatial envelope. *Intl. Journal of Computer Vision*, 42(3):145–175, 2001.

[78] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

[79] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of CSIIRW*, 2010.

[80] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[81] Roberto Perdisci and Andrea Lanzi. McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. *Computer Security Applications*, pages 301–310, December 2008.

[82] Jaishanker K. Pillai, Vishal M. Patel, Rama Chellappa, and Nalini K. Ratha. Secure and robust iris recognition using random projections and sparse representations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(9):1877–1893, 2011.

[83] PP Pullaperuma and AT Dharmaratne. Taxonomy of file fragments using gray-level co-occurrence matrices. In *Digital Image Computing: Techniques and Applications (DICTA), 2013 International Conference on*, pages 1–7. IEEE, 2013.

[84] K. Raman. Selecting features to classify malware. Technical report, 2012.

[85] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dussel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of DIMVA*, 2008.

[86] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[87] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 65–79. IEEE, 2012.

[88] Vassil Roussev and Simson L Garfinkel. File fragment classification-the case for specialized approaches. In *Systematic Approaches to Digital Forensic Engineering, 2009. SADFE'09. Fourth International IEEE Workshop on*, pages 3–14. IEEE, 2009.

[89] Vassil Roussev and Candice Quates. File fragment encoding classificationan empirical approach. *Digital Investigation*, 10:S69–S77, 2013.

[90] Phillipe Salembier and Thomas Sikora. *Introduction to MPEG-7: Multimedia Content Description Interface*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[91] I. Santos, Y. Penya, J. Devesa, and P. Bringas. N-grams-based file signatures for malware detection. In *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS*, pages 317–320, 2009.

[92] Igor Santos, Xabier Ugarte-Pedrero, Felix Brezo, Pablo Garcia Bringas, and José María Gómez Hidalgo. Noa: An information retrieval based malware detection system. *Computing and Informatics*, 32(1):145–174, 2013.

[93] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[94] Alexander K Seewald and Johannes Fürnkranz. An evaluation of grading classifiers. In *Advances in Intelligent Data Analysis*, pages 115–124. Springer, 2001.

[95] M. Shafiq, S. Tabish, F. Mirza, and M. Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2009.

[96] A. Stepan. Improving Proactive Detection of Packed Malware. 2006.

[97] Salvatore J Stolfo, Ke Wang, and Wei-Jen Li. Fileprints analysis for malware detection. In *Proceedings of WORMS 2005*, 2005.

[98] Kai Ming Ting and Ian H Witten. Stacked generalization: when does it work? 1997.

[99] Kai Ming Ting and Ian H Witten. Stacking bagged and dagged models. In *ICML*, pages 367–375, 1997.

[100] Kai Ming Ting and Ian H Witten. Issues in stacked generalization. *J. Artif. Intell. Res.(JAIR)*, 10:271–289, 1999.

[101] A. Torralba, R. Fergus, and W.T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1958–1970, 2008.

[102] A. Torralba, K.P. Murphy, W.T. Freeman, and M.A. Rubin. Context-based vision systems for place and object recognition. In *Proceedings of ICCV*, 2003.

[103] Andreas Töscher, Michael Jahrer, and Robert M Bell. The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, 2009.

[104] Joel A Tropp and Anna C Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *Information Theory, IEEE Transactions on*, 53(12):4655–4666, 2007.

[105] U.Bayer, C.Kruegel, and E.Kirda. Ttanalyze: A tool for analyzing malware. In *EICAR*, 2006.

[106] A. Walenstein, M. Hayes, and A. Lakhotia. Phylogenetic Comparisons of Malware. Virus Bulletin Conference, 2007.

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

[108] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy (Vol. 5, No. 2)*, March/April 2007.

[109] David H Wolpert. Combining generalizers using partitions of the learning set. *Nadel, L. and Stein, D., editors*, 1992.

[110] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[111] John Wright, Allen Y Yang, Arvind Ganesh, Shankar S Sastry, and Yi Ma. Robust face recognition via sparse representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(2):210–227, 2009.

[112] 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*, pages 95–104. ACM, 2010.

[113] Jian Zhang, Phil Porras, and Vinod Yegneswaran. Host-rx: Automated malware diagnosis based on probabilistic behavior models. Technical report, SRI International, 2009.

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