An Integrated Malware Detection and Classification System

by

Ronghua Tian
M.Eng., Chongqing University
B.Eng., Changchun University of Science and Technology

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

Deakin University

August, 2011
I am the author of the thesis entitled
An Integrated Malware Detection and Classification System
submitted for the degree of Doctor of Philosophy

This thesis may be made available for consultation, loan and limited copying in accordance with the
Copyright Act 1968.

'I certify that I am the student named below and that the information provided in the form is correct'

Full Name .......................... RONGHUA TIAN..........................
(Please Print)

Signed ..................................................................................

Date ..........................................................27/02/2012.................................
I certify that the thesis entitled

An Integrated Malware Detection and Classification System

submitted for the degree of

Doctor of Philosophy

is the result of my own work and that where reference is made to the work of others, due acknowledgment is given.

I also certify that any material in the thesis which has been accepted for a degree or diploma by any university or institution is identified in the text.

'I certify that I am the student named below and that the information provided in the form is correct'

Full Name .................................................. RONGHUA TIAN ..................................................
(Please Print)

Signed .................................................................

Date ................................................................. 27/02/2012 ..................................................
Publications

Submitted Journal Paper

Ronghua Tian, R. Islam, L. Batten, and S. Versteeg. “Robust classification of malware using a single test based on integrated static and dynamic features”

Refereed Conference Papers


Ronghua Tian, R. Islam, L. Batten, and S. Versteeg. “Differentiating malware from cleanware using behavioural analysis”. In Proceedings of 5th IEEE Inter-


Acknowledgments

I would like to express my great gratitude to all those who helped me during my research work and the writing of this thesis. This thesis could not be finished without the help and support of them.

My deepest gratitude goes first and foremost to my supervisor Prof. Lynn Margaret Batten for her guidance and assistance throughout my candidature. I appreciate her vast profound knowledge and skill in many areas. I am greatly indebted for her valuable instructions and suggestions in my research work as well as her careful reading of my thesis. Her kindness, wisdom, patience, strong passion, determination and diligence really impressed me. I have learnt from her a lot not only about academic studies, but also the professional ethics.

I would like to express my heartfelt gratitude to Dr. Steve Versteeg for his valuable ideas and suggestions in the academic studies, his constant encouragement and guidance during my research work and the writing of this thesis.

Also I gratefully acknowledge the help of Dr. Rafiqul Islam who offered considerable support during my research work. What’s more, I wish to extend my thanks to Dr. Lei Pan for his patient and meticulous guidance to the completion of this thesis. Thanks are also due to Miss Veelasha Moonsamy for her great encouragement and suggestions. I also owe a special debt of gratitude to all the professional experts from CA, including Tim Ebringer, Trevor Douglas Yann, David Wong and Shaun Faulds, who have instructed and helped me a lot during my research work.

Great thanks are due to all my postgraduate friends, who never failed to inspire me and give me great encouragement and suggestions.

I would also like to take this opportunity to thank the examiners for their time and devotion in reading my PhD thesis.
Last but not least, I would like to thank all my family members for the support they provided me all the way from the very beginning of my PhD research work. I must acknowledge my parents, my husband and my children, without whose unconditional love, encouragement and sacrifice, I would not have finished this thesis.
Contents

Acknowledgments iii

Abstract xx

Chapter 1: Introduction 1

1.1 Background ................................ 3

1.1.1 Definition of Malware ....................... 3

1.1.2 History of Malware ........................ 4

1.1.3 Type of Malware ........................... 6

1.1.3.1 Worms and Viruses ................... 6

1.1.3.2 Trojans ............................... 7

1.1.3.3 Rootkits ............................... 8

1.1.3.4 Backdoors ............................. 9

1.1.3.5 Spyware and Adware .................. 9

1.1.3.6 Bot ................................. 10

1.1.3.7 Hacker Utilities and other malicious programs .... 11

1.1.4 Naming Malware .......................... 11

1.2 Malware Detection and Classification .................. 14

1.2.1 The Proposal of this Research Problem ............... 14

1.2.2 Importance of the Research Problem ............... 16

1.2.3 Description of the Research Team ........................ 17

1.3 Outline of the Thesis .......................... 18
Chapter 2: Literature Review

2.1 General Flow of Signature-Based Malware Detection and Analysis

2.2 Related Work

2.2.1 Static Feature Extraction

2.2.2 Advantages and Disadvantages of Static Analysis

2.2.3 Dynamic (run-time) Feature Extraction

2.2.4 Advantages and Disadvantages of Dynamic Analysis

2.2.5 Machine Learning based Classification Decision Making Mechanisms

2.3 Our Proposed Method

2.3.1 Integrated Analysis and Extraction Approach

2.3.2 Machine Learning based Classification Decision Making Mechanisms

2.4 Hypotheses and Objective of System

2.5 Summary

Chapter 3: Architecture of the System

3.1 Introduction

3.2 Overview of System

3.3 Data Collection and Preprocess

3.3.1 Experimental Dataset

3.3.2 Static Preprocess

3.3.2.1 Unpacking

3.3.2.2 Reverse Engineering Ida2sql

3.3.3 Dynamic Preprocess

3.3.3.1 Virtual Machine Environment
Chapter 4: Function Length Features based Methodology

4.1 Introduction ........................................ 90
4.2 Related Work ........................................ 92
4.3 Data Preparation ...................................... 93
  4.3.1 IDA Disassembling Process ....................... 93
  4.3.2 IDA function ................................ 95
Chapter 5: String Features based Methodology 115

5.1 Introduction ........................................... 115
5.2 Related Work ........................................... 116
5.3 Data Preparation ........................................ 116
5.3.1 PSI .................................................. 116
5.3.2 Extract Printable String Information ................. 117
5.4 Experimental Set-up ..................................... 118
5.4.1 Motivation .......................................... 118
5.4.2 Test Dataset ......................................... 119
5.5 Printable String Information Test ......................... 119
5.5.1 Data Preprocessing and Feature Extraction .......... 121
Chapter 6: Combined Static Features based Methodology 132

6.1 Introduction ................................ 132
6.2 Related Work ................................ 133
6.3 Data Preparation ............................. 134
6.4 Experimental Set-up ........................... 135
   6.4.1 Motivation ............................. 135
   6.4.2 Test Dataset ........................... 135
6.5 Combined Test ................................. 136
   6.5.1 Overview of Experimental Process ...... 136
   6.5.2 Experimental Results .................... 137
6.6 Discussion .................................. 138
6.7 Summary .................................... 140

Chapter 7: Dynamic Methodology 142

7.1 Introduction ................................ 142
7.2 Related work ................................ 143
7.3 Data Preparation ............................. 144
   7.3.1 Dynamic Analysis Script ................. 145
7.4 Experimental Set-up ........................... 145
   7.4.1 Motivation ............................. 145
   7.4.2 Test Dataset ........................... 146
Chapter 8: Integrated Static and Dynamic Features

8.1 Introduction .................................................................................. 165
8.2 Related Work ................................................................................. 166
8.3 Data Preparation ........................................................................... 166
  8.3.1 FLF Vector ............................................................................. 166
  8.3.2 PSI Vector ............................................................................. 167
  8.3.3 Dynamic Vector ..................................................................... 168
  8.3.4 Integrated Vector .................................................................. 169
8.4 Experimental Set-up ...................................................................... 170
  8.4.1 Motivation ............................................................................. 170
  8.4.2 Test Dataset .......................................................................... 172
8.5 Integrated Tests ............................................................................. 172
  8.5.1 Family Classification ............................................................ 174
  8.5.2 Malware Versus Cleanware Classification .............................. 174
  8.5.3 Using the Integrated Method on Old and New Families ......... 179
Appendix B: Experimental Dataset

<table>
<thead>
<tr>
<th>Section</th>
<th>Dataset Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>Adclicker</td>
<td>226</td>
</tr>
<tr>
<td>B.2</td>
<td>Bancos</td>
<td>227</td>
</tr>
<tr>
<td>B.3</td>
<td>Banker</td>
<td>227</td>
</tr>
<tr>
<td>B.4</td>
<td>Gamepass</td>
<td>227</td>
</tr>
<tr>
<td>B.5</td>
<td>SillyD1</td>
<td>229</td>
</tr>
<tr>
<td>B.6</td>
<td>Vundo</td>
<td>229</td>
</tr>
<tr>
<td>B.7</td>
<td>Frethog</td>
<td>231</td>
</tr>
<tr>
<td>B.8</td>
<td>SillyAutorun</td>
<td>231</td>
</tr>
<tr>
<td>B.9</td>
<td>Alureon</td>
<td>232</td>
</tr>
<tr>
<td>B.10</td>
<td>Bambo</td>
<td>232</td>
</tr>
<tr>
<td>B.11</td>
<td>Boxed</td>
<td>233</td>
</tr>
<tr>
<td>B.12</td>
<td>Clagger</td>
<td>234</td>
</tr>
<tr>
<td>B.13</td>
<td>Robknot</td>
<td>235</td>
</tr>
<tr>
<td>B.14</td>
<td>Robzips</td>
<td>237</td>
</tr>
<tr>
<td>B.15</td>
<td>Looked</td>
<td>240</td>
</tr>
<tr>
<td>B.16</td>
<td>Emerleox</td>
<td>241</td>
</tr>
<tr>
<td>B.17</td>
<td>Agobot</td>
<td>242</td>
</tr>
</tbody>
</table>

Appendix C: Function Length Extraction Procedure

<table>
<thead>
<tr>
<th>Section</th>
<th>Procedure Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>Database Stored Procedure: Fun_Len_Family</td>
<td>244</td>
</tr>
<tr>
<td>C.2</td>
<td>Database Stored Procedure: Fun_Len_Module</td>
<td>245</td>
</tr>
<tr>
<td>C.3</td>
<td>Database Internal Function: GetFun</td>
<td>246</td>
</tr>
<tr>
<td>C.4</td>
<td>Database Internal Function: GetBasicData</td>
<td>247</td>
</tr>
</tbody>
</table>

Appendix D: Ida2DB schema

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Basic Tables</td>
<td>248</td>
</tr>
</tbody>
</table>
List of Figures

2.1 General Flow of Signature-based Malware Detection and Analysis .... 22
3.1 Architecture of Our Malware Detection and Classification System .... 47
3.2 Implementation of Our Malware Detection and Classification System ... 48
3.3 Number of New Malicious Programs Detected by Kaspersky Lab in 2006 and 2007 [Gos08] .................................................. 50
3.4 Distribution of Malicious Programs in 2006 [Gos08] ...................... 50
3.5 Distribution of Malicious Programs in 2007 [Gos08] ...................... 50
3.6 VMUnpacker 1.3 .......................................................... 54
3.7 Compilation and the Reverse-engineering Process ......................... 55
3.8 The Interface of AllEnOne .............................................. 58
3.9 Dynamic Analysis Preprocess .......................................... 60
3.10 Detours: Logical Flow of Control for Function Invocation with and without Interception [HB99] .......................................... 61
3.11 Trace Tool HookMe ........................................................ 62
3.12 Idb2DBMS Schema ...................................................... 67
3.13 An Example of a Log File of API Calls .................................. 70
3.14 Performance of Classification ........................................... 87
4.1 IDA Disassembling Process .............................................. 93
4.2 IDA Function Data ........................................................ 96
4.3 Related Database Programs .............................................. 97
4.4 Function Fetch ............................................................. 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5 Function Length Pattern Samples from Robzips Family</td>
<td>99</td>
</tr>
<tr>
<td>4.6 Function Length Pattern Samples from Robknot Family</td>
<td>100</td>
</tr>
<tr>
<td>4.7 Overview of Our Experimental Process</td>
<td>102</td>
</tr>
<tr>
<td>5.1 Exporting of PSI</td>
<td>117</td>
</tr>
<tr>
<td>5.2 Strings Window in IDA</td>
<td>118</td>
</tr>
<tr>
<td>5.3 Overview of PSI Experiment</td>
<td>122</td>
</tr>
<tr>
<td>5.4 Global String List</td>
<td>123</td>
</tr>
<tr>
<td>5.5 Comparison of Classification Accuracy (with and without Boosting)</td>
<td>127</td>
</tr>
<tr>
<td>6.1 Combined Feature Vector Example</td>
<td>134</td>
</tr>
<tr>
<td>6.2 Combined Static Features Based Classification Process</td>
<td>137</td>
</tr>
<tr>
<td>6.3 Comparison of Classification (with and without boosting)</td>
<td>139</td>
</tr>
<tr>
<td>6.4 Comparison with PSI method</td>
<td>140</td>
</tr>
<tr>
<td>7.1 Overview of Dynamic Feature Based Experiment</td>
<td>147</td>
</tr>
<tr>
<td>7.2 Sample Feature Sets of A Malware File</td>
<td>149</td>
</tr>
<tr>
<td>7.3 Comparison of Average Classification Accuracy Between Base and</td>
<td></td>
</tr>
<tr>
<td>Meta Classifier.</td>
<td>155</td>
</tr>
<tr>
<td>7.4 Comparison of Family Classification Results</td>
<td>158</td>
</tr>
<tr>
<td>8.1 Example of an FLF Bin Distribution</td>
<td>167</td>
</tr>
<tr>
<td>8.2 Example of Data Used in a PSI Vector</td>
<td>168</td>
</tr>
<tr>
<td>8.3 Example of Data Used in a Dynamic Feature Vector</td>
<td>169</td>
</tr>
<tr>
<td>8.4 Integrated Feature Extraction Model</td>
<td>170</td>
</tr>
<tr>
<td>8.5 Data Used in Generating an Integrated Feature Vector</td>
<td>171</td>
</tr>
<tr>
<td>8.6 The General Classification Process</td>
<td>173</td>
</tr>
<tr>
<td>8.7 Compare FP Rate of Old and Integrated Methods (Meta-classifier)</td>
<td>186</td>
</tr>
<tr>
<td>8.8 Compare FN of Old and Integrated Methods (Meta-classifier)</td>
<td>186</td>
</tr>
</tbody>
</table>
8.9 Compare Accuracy of Old and Integrated Methods (Meta-classifier) 187
8.10 Compare FPRate of Old and New Malware Families Using Integrated Method. 191
8.11 Compare FNRate of Old and New Malware Families Using Integrated Method. 191
8.12 Compare Accuracy of Old and New Malware Families Using Integrated Method. 192

A.1 Related Tables 217
A.2 Experiment Results 222
A.3 Execution Time Trend 224

B.1 Text File Created by Robzips 238
B.2 Messages Displayed on a Console Window by Robzips 238
B.3 Messages Displayed on a Console Window by Robzips 239

C.1 Function Length Feature Extraction 244
List of Tables

3.1 Experimental Set of 2939 Files .......................... 52
3.2 Fetch Basic Information of Executable Files from a Specific Family . 65
3.3 Fetch Main Function Information of Some Assigned Executable Files 66
3.4 Fetch the List of Instructions of a Specific Basic Block from a File . 66
3.5 Fetch the Specific Instruction .................................. 68
3.6 Fetch All the Printable String Information from a File ............ 69

4.1 Experimental Set of 721 Malware Files ...................... 101
4.2 Function Length Frequency Results .......................... 106
4.3 Function Length Pattern Results .............................. 111
4.4 Running Times in the Function Length Experiments .............. 111

5.1 Experimental Set of 1367 Files .............................. 120
5.2 Average Family Classification Results in PSI Experiment ...... 126
5.3 Weighted Average Family Classification Results in PSI Experiment . 127
5.4 Malware Versus Cleanware Results in PSI Experiment ........ 128
5.5 Running Times in the Printable String Information Experiments . 129
5.6 Comparison of Our Method with Existing Work ............... 131

6.1 Classification Results for Base Classifier .................... 137
6.2 Classification Results for Meta classifier .................... 138
6.3 Comparison of Our method with Similar Existing Work .......... 141

7.1 Example Global Frequencies and File Frequencies ............... 150
Abstract

With the rise in the shadow Internet economy, malware has developed into one of the major threats to computers and information systems throughout the world. Antivirus analysis and detection is a major resource in maintaining an organization’s antivirus preparedness and responsiveness during malware outbreak events, thereby contributing to the well-being of its IT health, and consequently to that of the economy as a whole.

Currently the majority of anti-virus detection systems are signature-based, which means that they try to identify malware based on a single feature. The major disadvantage of such signature-based detection systems is that they cannot detect unknown malware, but only identify variants of malware that have been previously identified. Moreover more and more malware writers use obfuscation technology such as packing, encrypting or polymorphisms, to avoid being detected by antivirus detection engines. With a signature-based approach, frequent and recurrent updates of the malware signature database are imperative as huge numbers of malware variants are released every day. Therefore the traditional signature-based detection system is neither efficient nor effective in defeating malware threats.

In the search for effective and efficient solutions to the malware problem, researchers have moved away from the signature approach and the new detection and classification methods can basically be categorized into two types: static methods and dynamic methods. In the static method, researchers acquire useful information from static disassembling code; in the dynamic method, they use information from runtime trace reports of executable files.

Earlier studies in malware detection and classification focused on calculating similarity between two pieces of code by using program comparison technologies. Most program comparison technologies require the computation of the distances between all
pairs of files, which invariably results in a computational complexity of $O(n^2)$. These technologies are effective in the early stage of malware development. But given the rapidly increasing number of malware released every day, these technologies should be replaced by more scalable and effective methodologies.

When I started the research work in 2007, relatively little work had been done on the study of improving the detection and classification accuracy by using machine learning methods. And there was little published work dealing with the problem of malware detection which is the problem of distinguishing between malware and cleanware. From the point of view of performance, the best malware classification accuracy at that time was 91.6% as mentioned in the literature review presented in Chapter 2 of my thesis.

The aims of this thesis are to develop effective and efficient methodologies which can be applied to continuously improve the performance of detection and classification on malware collected over an extended period of time. And the target of such a system is 97% malware detection and classification accuracy, which therefore significantly improves on current work.

In developing the thesis, we test a set of hypotheses including the idea that combining different features or characteristics of a malware file in the analysis may be more effective in classification than a single feature. This is because malware writers attempt to avoid detection by obfuscation of some features. We show that indeed this hypothesis is valid by using a combination of static and dynamic features in developing an integrated test which is considerably more effective than the any of the tests based on these features individually.

One of the claims in the research literature is that, over time, malware becomes resistant to the older anti-virus detection methods. We demonstrate the robustness of our integrated feature method by testing it on malware collected over 2003-2008
and malware collected over 2009-2010. The results indicate that it is indeed possible to bypass obfuscation techniques by using a combination of several features.
Chapter 1

Introduction

Malware, short for malicious software, is a relatively new concept, but is not a new research field. Intrinsically malware is a variety of hostile, intrusive, or annoying software or program code designed to secretly access a computer system without the owner’s informed consent. In a sense, the development of malware is closely related to the development of software engineering. Since software engineering was first officially proposed at the 1968 NATO Software Engineering Conference [Wir08], the art of programming has evolved into a profession concerned with how best to maximize the quality of software and how to create it. From its very beginning in the 1960s, malware has also evolved into the most significant threat to computer network systems, especially in the last three decades. Along with the growth of the Internet, there has been a dramatic growth in instances of malware in recent years [YWL+08]. According to the 2010 Annual Report from PandaLabs [Pan11], “Rounding up the figures for 2010, some 20 million new strains of malware have been created (including new threats and variants of existing families), the same amount as in the whole of 2009. The average number of new threats created every day has risen from 55,000 to 63,000.” One of the major threats on the Internet today is malware and the underlying root cause of most Internet security problems is malware.
Malware has already become a global problem which has affected different parts of the world in different ways. According to the Microsoft security intelligence report [Mic09] in 2009, threat assessments for 26 individual locations, including Australia, Canada, China and the United States, shows that the vast majority of threats detected on infected computers come from malware. For example, in the United States, malware is the main security threat, which accounted for 72.9 percent of all threats detected on infected computers. In Australia, this number was 76.2 percent.

With the rise in the shadow Internet economy, malware is no longer simply used to damage, break or intrude on computer network systems, but now exists primarily as a tool used by criminals to make a profit. As [Sch07] states the teenagers who wrote viruses have grown up and now they’re trying to make money. The shadow Internet economy is worth over $105 billion. Online crime is bigger than the global drugs trade [Sch07].

Malware makers are often looking for one-time development of specific code to generate new variants of existing malware, instead of developing new malware from scratch. In this case, variants of existing malware can be developed easily and quickly, and therefore, can be rapidly brought to market in the shadow economy. According to statistical analysis of Microsoft Security Intelligence Report [BWM06], of the 97,924 variants collected in the first half of 2006, the top seven families accounted for more that 50 percent of all variants found, and the top 25 families accounted for over 75 percent. This means there is a very big opportunity that any new malicious program found in the wild is a variation of a previous program.

Thus, there is a need to develop an automatic malware detection and classification system to identify the variants of existing malware, in order to guide analysts in the selection of samples that require the most attention. Over the last decade, researchers have adopted a variety of solutions in order to control malware. Much research has
been conducted on developing automatic malware detection and classification systems using static or dynamic analysis methods.

This thesis aims to develop effective and efficient methodologies which can be used to continuously improve the performance of detecting and classifying malware collected over an extended period of time.

In this chapter, I present the background information of my research topic by describing the definition of malware, malware history, malware naming, and types of malware. Following this I state my research problem and challenge in Section 1.2. Section 1.3 provides an outline of the remainder of the thesis, and Section 1.4 summarizes the chapter.

1.1 Background

In this section I comprehensively profile malware by presenting a definition of malware, and malware history. I also identify and name different types of malware.

1.1.1 Definition of Malware

Malware is short for “malicious software”, and refers to software programs designed to damage or perform other unwanted actions on a computer system. In Spanish, “mal” is a prefix that means “bad”, therefore malware means “badware”.

Many people have tried to define malware by describing its essential characteristics. As early as 1986, Fred Cohen presented the first rigorous mathematical definition for a computer virus in his Ph.D thesis [Coh85]. He wrote “A virus can be described by a sequence of symbols which is able, when interpreted in a suitable environment (a machine), to modify other sequences of symbols in that environment by including
a, possibly evolved, copy of itself.” Although his thesis only focused on viruses and did not consider the more general issue of malware, he coined the term ‘virus’ which was a fundamental distinction. On November 10, 1983, at Lehigh University, Cohen demonstrated a virus-like program on a VAX11/750 system. The program was able to install itself to, or infect, other system objects. This is the birth of experimental computer virus.

The definition of malware varies as the development of the computer system, the Internet and malware continue. Software is considered malware based on the perceived intent of the creator rather than on any particular features. So from a practical point of view, the following definition is popularly accepted: Malware is software designed to infiltrate or damage a computer system without the owner’s informed consent. It is a combination of the words “malicious” and “software” and the expression is a general term used by computer professionals to mean a variety of hostile, intrusive, or annoying software or program code.

1.1.2 History of Malware

Every field of study has its own unique developing history. We cannot understand malware without first understanding its history. By understanding the history of Malware, and learning the names and significant events that have shaped the development of this field of study, we are able to better understand references from experts in this field. Following is a brief description of the history of malware.

With the emergence of computers, malware became increasingly common. As early as 1949, computer pioneer John von Neumann presumed that a computer program could reproduce, which is the most primitive conceptual description of malware. In [VN68], he deduced that we could construct automata which could reproduce themselves and, in addition, construct others. It is generally accepted that the first
malware was a virus called the Creeper, which infected ARPANET, the forerunner of the modern Internet, in 1971. It was created by engineer Bob Thomas, working for BBN. The Creeper was not, however, malicious. Infected machines would simply display the message, “I’m the creeper: catch me if you can,” but they did not suffer any lasting damage. As a direct response to the Creeper challenge, the first piece of anti-virus software, the Reaper was created, which was also a self-replicating program that spread through the system in much the same way the Creeper did. The Reaper removed the offending virus from infected computers, and just as quickly as it had spread, the Creeper was caught.

Before the wide spread of the Internet, most communication networks were limited by only allowing communications between stations on the local network, therefore the earlier prevalence of malware was rather limited. As the Internet evolved, so did the nature of the threat. It is not surprising then, the evolution of malware is directly related to the success and evolution of the Internet.

According to the white paper from McAfee [McA05], one of the earliest viruses named Brain was introduced in 1986, infecting the boot sector of floppy disks, which was the principal method of transmitting files of data from one computer to another. The first major mutation of viruses took place in July 1995. This was when the first macro virus was developed. This virus was notably different from boot sector viruses because it was written in a readable format.

The next major mutation of viruses took place in 1999 when a macro-virus author turned his attention to the use of e-mail as a distribution mechanism. This saw the birth of Melissa, the first infamous global virus. After Melissa, viruses were no longer

\footnote{Christopher Koch presents the history of malware in a cyber crime timeline in CIO (Chief Information Officer) Magazine [Koc07], it shows that the earliest malware appeared between 1970s-80s.}
solely reliant on file sharing by floppy disk, network shared files, or e-mail attachments.Viruses had the capability to propagate through Internet applications.

Malware has since evolved into many types, such as viruses, worms, Trojan horses, backdoors, and rootkits, with these new threats identifying and preying upon vulnerabilities in applications and software programs to transmit and spread attacks. In 2002, these threats began to combine, and the blended threat was born. By utilizing multiple techniques, blended threats can spread far quicker than conventional threats.

1.1.3 Type of Malware

With the rapid development and popularity of the Internet, malware has become more and more complicated and has from the very first virus to worms, and Trojans, and now the currently notorious rootkits. In this sub-section I attempt to clarify the meaning of each of these terms to develop an understanding of what they are and their potential dangers.

1.1.3.1 Worms and Viruses

The earliest and best known types of malware are worms and viruses. Worms include programs that propagate via LANs or the Internet with malicious objectives, including penetrating remote machines, launching copies on victim machines and further spreading to new machines. Worms use different networking systems to propagate, such as email, instant messaging, file-sharing (P2P), IRC channels, LANs and WANs.

Most existing worms spread as files in one form or another, including email attachments, ICQ or IRC messages and accessible files via P2P networks. There are a small number of so-called fileless or packet worms which spread as network packets and
directly penetrate the RAM of the victim machine, where the code is then executed. Worms use a variety of exploits for penetrating victim machines and subsequently executing code, and these exploits may include emails that encourage recipients to open an attachment, poorly configured networks, networks that leave local machines open to access from outside the network or vulnerabilities in an operating system and its applications.

Viruses cover programs that spread copies of themselves throughout a single machine in order to launch and/or execute code once a user fulfills a designated action, and it also penetrates other resources within the victim machine. Unlike worms, viruses do not use network resources to penetrate other machines. Copies of viruses can penetrate other machines only if an infected object is accessed and the code is launched by a user on an uninfected machine. This can happen in the following ways: the virus infects files on a network resource that other users can access; the virus infects removable storage media which are then used in a clean machine; or, the user attaches an infected file to an email and sends it to a “healthy” recipient. Viruses are sometimes carried by worms as additional payloads or they themselves can include backdoor or Trojan functionality which destroys data on an infected machine.

A virus requires user intervention to spread, whereas a worm spreads automatically. Because of this distinction, infections transmitted by email or Microsoft Word documents, which rely on the recipient opening a file or email to infect the system, would be classified as a virus rather than a worm.

1.1.3.2 Trojans

Trojan is short for “trojan horse” and is derived from the Greek myth of the Trojan War.

Trojan is a hidden program which secretly runs commands in order to accomplish
its goals without being shut down, or deleted by the user or administrator of the computer on which it is running. Trojan appears to perform a certain action but in fact performs another similar to a computer virus. Contrary to popular belief, this action, usually encoded in a hidden payload, may or may not actually be malicious. Trojan horses are currently notorious for their use in the installation of backdoor programs. A trojan, known as dropper, is used to begin a worm outbreak by injecting the worm into users’ local networks.

This type of malware includes a wide variety of programs that perform actions without the user’s knowledge or consent, including the collection of data and sending it to a cyber criminal, destroying or altering data with malicious intent, causing the computer to malfunction, or using a machine’s capabilities for malicious or criminal purposes, such as sending spam.

Broadly speaking, a Trojan is any program that invites the user to run it, concealing a harmful or malicious payload. The payload may take effect immediately and can lead to many undesirable effects, such as deleting the user’s files or further installing malicious or undesirable software.

### 1.1.3.3 Rootkits

A “rootkit” is a program (or combination of several programs) designed to take fundamental control (in Unix terms “root” access, in Windows terms “Administrator” access) of a computer system, without authorization by the system’s owners or legitimate managers. Access to the hardware (i.e, the reset switch) is rarely required as a rootkit is intended to seize control of the operating system running on the hardware. Typically, rootkits act to obscure their presence on the system through subversion or evasion of standard operating system security mechanisms. Often, they are also Trojans as well, thus fooling users into believing they are safe to run on their sys-
tems. Techniques used to accomplish this can include concealing running processes from monitoring programs, or hiding files or system data from the operating system.

Rootkits may have originated as regular, though emergency applications intended to take control of an unresponsive system, however in recent years they have mostly been malware to help intruders gain access to systems while avoiding detection. Rootkits exist for a variety of operating systems, such as Microsoft Windows, Mac OS X, Linux and Solaris. Rootkits often modify parts of the operating system or install themselves as drivers or kernel modules, depending on the internal details of an operating system’s mechanisms.

1.1.3.4 Backdoors

Backdoor is a method of bypassing normal authentication procedures. Once a system has been compromised (by one of the above methods, or in some other way), one or more backdoors may be installed in order to allow easier access in the future. Backdoors may also be installed prior to malicious software, to allow attackers entry. It has often been suggested that computer manufacturers preinstall backdoors on their systems to provide technical support for customers, but this has never been reliably verified. Crackers typically use backdoors to secure remote access to a computer, while attempting to remain hidden from casual inspection. To install backdoors, crackers may use Trojan horses, worms, or other methods.

1.1.3.5 Spyware and Adware

Spyware is any software installed on the system without the owner’s knowledge. Spyware collects information and sends that information back to the attacker so the attacker can use the stolen information in some nefarious way, to learn and steal passwords or credit card numbers, change the settings of your browser, or add abom-
nable browser toolbars. A trojan horse is one of the most common ways spyware is distributed and is usually bundled with a piece of desirable software that the user downloads from the Internet. When the user installs the software, the spyware is also installed. Spyware authors who attempt to act in a legal fashion may include an end-user license agreement that states the behavior of the spyware in loose terms, which the users are unlikely to read or understand.

1.1.3.6 Bot

“Bot” is short for the word “robot”, which is another type of malware and is an automated process that interacts with other network services. A typical use for bots is to gather information (such as web crawlers), or interact automatically with instant messaging (IM), Internet Relay Chat (IRC), or other web interfaces. Bot software enables an operator to remotely control each system and group them together to form what is commonly referred to as a zombie army or botnet [BCJ+09, CJM05]. Attackers use these zombies or bots as anonymous proxies to hide their real identities and amplify their attacks.

A Botnet is a large pool of compromised computer hosts across the Internet. Attackers can use a botnet to launch broad-based, remote-control, flood-type attacks against their targets. Currently the bots found in the wild are a hybrid of previous threats. This means they may propagate like worms, hide from detection like many viruses, attack like many stand-alone tools, and have an integrated command and control system. They have also been known to exploit back doors opened by worms and viruses, which allows them access to controlled networks. Bots try to hide themselves as much as they can and infect networks in a way that avoids immediate notice.
1.1.3.7 Hacker Utilities and other malicious programs

Hacker utilities and other malicious programs include:

- Utilities such as constructors that can be used to create viruses, worms and Trojans.
- Program libraries specially developed to be used in creating malware.
- Hacker utilities that encrypt infected files to hide them from antivirus software.
- Jokes that interfere with normal computer function.
- Programs that deliberately misinform users about their actions in the system.
- Other programs that are designed to directly or indirectly damage local or networked machines.

Functionality and infected methods of current malware have become more and more complicated and diverse. Current malware is often a composite creation and does not easily fit into the above categories. Instances of malware always combine several approaches or technologies in order to avoid being detected by an anti-virus engine. For instance, worms now often include trojan functions by containing a payload which installs a back door or bot to allow remote access and control. They are no longer purely worms, but hybrid malware instances that include all the functionality of a virus, worm, trojan and/or spyware together.

1.1.4 Naming Malware

Many security vendors use naming conventions based on the CARO (Computer AntiVirus Research Organization) naming scheme with minor variations. The CARO malware naming scheme was created almost 20 years ago, and to date, it remains
the naming scheme most widely used in anti-virus products. CARO is an informal organization, and is composed of a group of individuals who have been working together since around 1990 across corporate and academic borders to study computer malware. At a CARO meeting in 1991, a committee was formed with the objective of reducing the confusion in naming viruses. This committee decided that a fundamental principle behind the naming scheme should be that malware should be grouped into families according to the similarity of its programming code. They proposed and published a naming convention which is the rudiment of current adopted naming schema. This naming schema revised in 2002 and is constantly kept up to date in order to reflect any future modifications. [Bon05]. The full name of malware consists of up to eight parts, separated points (.). The general format of a Full CARO Malware Name is:

```
[type://]<platform>/<family>.[<group>].[<length>].<variant>
[<modifiers>][!<comment>]
```

where the items in square brackets are optional. According to this format, only the family name and the variant name of a piece of malware are mandatory and even the variant name can be omitted when it is reported.

- The type part indicates the type of malware it is and the naming scheme permits the following different types: virus, dropper, intended, Trojan, pws, dialer, backdoor, exploit, tool or garbage. Currently, these malware types are the only malware types permitted by the CARO Malware Naming Scheme. Notably, there is no special malware type for a worm, with the reason being that it seems impossible to reach an agreement among anti-virus researchers on what exactly a worm is. In order to avoid confusion, this naming scheme does not use such
a malware type at all. While the anti-virus producer may put this information in the comment field if they absolutely have to report that something is a worm. In addition, there are no malware types for spam, adware, spyware, phishing scams, non-malicious applications or unwanted applications, however these malware types may be introduced in the future. Currently, some anti-virus vendors have chosen to report such things with their products.

- The platform part specifies the platform on which the malware works. This can be an operating system (e.g., “PalmOS”), a set of operating systems (e.g., “Win32”), an application (e.g., “ExcelMacro”), or a language interpreter (e.g., “VBS”) or a file type.

- The family name is the only part that a virus scanner uses to detect the malware. This is due to one of the fundamental principles that malware should be grouped into families according to the similarity of its code in the Malware Naming Scheme. This is useful for developers of anti-virus software because malware that is programmed in a similar way usually needs similar methods of detection and removal.

- The group part is used when a large subset of a malware family contains members that are sufficiently similar to each other and sufficiently different from the other members of the same family, yet at the same time the members of this subset are not similar enough to each other to be classified as variants.

- The length part indicates the infective length of the particular piece of malware.

- The variant part is used to distinguish between different malware programs that belong to the same family.

- The modifier part lists some properties of the malware that are deemed important enough to be conveyed to the user immediately.
• The comment part is used to report a malware that is not included in this scheme.

Although this schema is widely used in anti-virus products, there is no product that is absolutely compliant with the schema. Different anti-virus vendors customize it according to their practical requirements, and family and variant names for the same malware could differ between vendors, however in general, the variations are minor and people can find out more by reading the detailed description.

1.2 Malware Detection and Classification

1.2.1 The Proposal of this Research Problem

Anti-malware analysis and detection system is a major resource in maintaining an organization’s antivirus preparedness and responsiveness during outbreak events. This preparation and response contributes to the well-being of the organizations IT health, and consequently to the economy as a whole. Nevertheless, the use of such software is predicated on an initial and accurate identification of malware that is used to develop methods to aid the automation of malware identification and classification.

Malware identification and analysis is a technically intense discipline, requiring deep knowledge of hardware, operating systems, compilers and programming languages. To compound the problem, successful identification and analysis by malware analysts has been confounded by the use of obfuscated code in recent years. Malware writers have adopted obfuscation technology to disguise the malware program so that its malicious intent is difficult to detect. Obfuscation techniques can involve obscuring a program’s behavioral patterns or assembly code [Mas04, SXCM04], encrypting some components, or compressing some of the malware data thereby de-
destroying the detectable code patterns. There are freely available open-source and 
commercial obfuscation tools which purport to harden applications against piracy 
and de-obfuscation techniques. In the open source project UPX, for example, 
(http://upx.sourceforge.net), the obfuscation is designed to be reversible, but hackers 
make slight alterations to the source code to destroy this property. Manual unpacking 
by an experienced person can still be done in this instance, but an automated process 
becomes extremely difficult.

In the anti-virus industry, the majority of anti-virus detection systems are 
signature-based which was effective in the early stages of malware. But given the 
rapid development of malware technologies and the huge amount of malware released 
every day, the signature-based approach is neither efficient nor effective in defeating 
malware threats. Currently, we are in need of finding more effective and efficient 
approaches.

To deal with the rapid development of malware, researchers have shifted from 
a signature-based method, to new approaches based on either a static or dynamic 
analysis to detect and classify malware. In the static analysis, researchers focus on 
disassembling code to acquire useful information to represent malware, whereas in the 
dynamic analysis, they monitor the execution of malware in a controlled environment 
and extract information from runtime trace reports to represent the malware. In 
using both static and dynamic methods, it is crucial to find key features that can 
represent malware and that are effective in malware detection and classification.

One of the complaints about malware detection and classification is that once the 
method is made public, a malware writer need only obfuscate the principal feature 
used in the classification to avoid detection. In developing this thesis, I designed and 
implemented several experiments based on static and dynamic methodologies. The 
results of these experiments led me and my research colleagues to believe that an
integrated method could be developed that incorporates static and dynamic methods to complement each other to make the detection and classification effective and robust to changes in malware evolution.

A second complaint about standard methods of classification is that they are based on a given set of malware and may apply to that set, but may not fare as well on more recent or future malware. In developing this thesis, we introduced more recent malware families into our experiments to test the robustness of our method.

1.2.2 Importance of the Research Problem

The significance of an automatic and effective malware detection and classification system has the following benefits:

- When new malware is found in the wild, it can quickly be determined whether it is a new instance of malware or a variant of known family.

- If it is a variant of known family, the anti-virus analysts can predict the possible damage it can cause, and can launch the necessary procedure to quarantine or remove the malware. Furthermore, given sets of malware samples that belong to different malware families, it becomes significantly easier to derive generalized signatures, implement removal procedures, and create new mitigation strategies that work for a whole class of programs [BCH+09].

- Alternatively, if it is new malware, the system can still detect the similarity between the new malware and other known malware which provides valuable information for further analysis.

- Analysts can be free from the grueling analysis of a huge number of variants of known malware families and focus on the truly new ones.
1.2.3 Description of the Research Team

The research work presented in this thesis is part of a broader project - “Analysis and Classification of Malicious Code” - which is supported by ARC grant number LP0776260 under the auspices of the Australian Research Council and by research partner CA Technologies. During the course of this project I have worked closely with my supervisor Professor Lynn Batten from Deakin University, Dr. Steve Versteeg from CA Technologies, and Dr. Rafiqul Islam from Deakin University.

In the initial stages of the project, my role was to unpack malware from the CA Zoo and set up a database to collect data from the malware. I also customized the ida2sql so that it would manage the data in the format we wished to use for the project.

Based on the data gathered, I performed preliminary testing to determine if any of the extracted malware features might be useful in classification. I discovered that we were able to distinguish between malware based on function length features. I used this discovery as the basis of the function length tests discussed in Chapter 4. In the next stage of my work, I added more malware samples and used string features as a basis for testing and then compared the results with those for function length. (Chapter 5.)

This led me to consider using the dynamic information from the malware (as performed by other researchers) to see how the results compared with those on the static features I had used. At this point, the extended team became interested in malware detection rather than classification, and this led to the inclusion of cleanware in the testing. The cleanware was at first treated as another family, but I developed a test for the dynamic research which tested all the malware against the cleanware. This is important because a significant amount of cleanware - such as auto-updating, uses the same APIs which are commonly exploited by malware, therefore, naive approaches
can incorrectly identify these cleanware programs as malware. My team finally wanted to derive an integrated test using both static and dynamic features. To set this up, I had to determine a common data set which had usable log input for both tests. This meant I had to unpack more malware, find more cleanware and rerun the previous tests all over again. At this point, I had to make some decisions about how to derive a common sized vector to incorporate all the features we needed to include. The integrated dynamic test results are shown in Chapter 8. We achieved our target of (over) 97% accuracy with this final test.

1.3 Outline of the Thesis

The thesis comprises nine chapters, and is organized as follows:

**Chapter 2** provides the literature review of my research topic. In this Chapter, I begin with an understanding of the general flow of traditional Signature-based Malware Detection and Analysis. Following this, I present two core problems of malware detection and classification, then I present a review of the literature that incorporates static and dynamic methods, with discussion on the merits and limitations of these methods. To conclude this chapter, our proposed method is presented.

**Chapter 3** introduces the architecture of our malware detection and classification system. The system is separated into different layers according to the flow of process. It includes a Data Collection and Data Preprocess Layer, a Data Storage Layer, an Extraction and Representation Layer, a Classification Process Layer, and a Performance Assessment Layer. I provide an explanation for each of these layers.
Chapter 4 presents a simple and scalable method which is based on function length features extraction and representation. Function length frequency and the function length pattern are investigated.

Chapter 5 presents another static feature extraction and presentation method, focusing on the PSI (Printable String Information).

Chapter 6 presents a combined static approach. FLF (Function Length Frequency) and PSI (Printable String Information) static features are combined to improve the performance of detection and classification.

Chapter 7 proposes a scalable approach for detecting and classifying malware by investigating the behavioral features using a dynamic analysis of malware binaries.

Chapter 8 integrates both static and dynamic approaches to detect and classify the malware in order to improve robustness and performance.

Chapter 9 presents the conclusion to this thesis.

1.4 Summary

In this chapter, I described the background information related to my research, proposed my research problem, explained the importance of this research and stated the aim of my research.
Chapter 2

Literature Review

In this chapter, I present a literature review on my topic of research. First I describe the traditional method used by the anti-virus industry, and in Section 2.2 I present literature related to malware analysis and extraction approaches as well as the malware classification decision making mechanism. In Section 2.3 I explain our proposed methodologies for malware detection and classification.

2.1 General Flow of Signature-Based Malware Detection and Analysis

Currently, the majority of anti-virus detection systems are signature-based. A signature is a distinctive piece of code within an executable file which is used to identify the file and which is expressed in terms of byte sequences or instruction sequences. Executable files are disassembled by Reverse Engineering Software, and then anti-virus engineers examine the disassembled code to identify distinctive signatures manually. Signatures are then stored in a signature database or signature repository.
Signature-based detection involves searching the signature databases for the matched pattern.

Figure 2.1 shows the general procedure for signature-based detection. When a executable file arrives, it is sent to the anti-virus engine. The anti-virus engine initially checks if the sample is packed or not; if it is packed then the engine unpacks it and passes the unpacked version to a specific scanner of an anti-virus engine. A scanner is a specific part of the engine which deals with a specific file type. Executable unpacking is a feature built into an anti-virus engine and it uses the creation of new scanning streams to write out an executable that is an unpacked version of the currently scanned executable. A given scanner may use a specialized searching algorithm and behaves in ways suitable to its file type. For instance, a Win32 scanner processes Windows Portable Executable (PE) files; whereas a DOS scanner is used for scanning MZ executables files, DOS COM files and any other binary files which are not recognized by other scanners. The anti-virus engine then evaluates the disassembling code of the file with the malware signature database by comparing specific bytes of code against information in its malware signature database. If the file contains a pattern or signature that exists within the database, it is deemed malicious. A report is then generated giving detailed information and the anti-virus engine either quarantines or deletes the file, depending upon the anti-virus engine configurations. If no matched signature is found in that file, then the suspicious file is passed on to anti-virus engineers for further analysis with the malware signature database updated.

The disadvantage of such a signature-based detection system is that it cannot detect unknown malware. Since signatures are created by examining known malware, the detection can only detect “known malware”. A signature-based detection system cannot always detect variants of known malware. Therefore, signature-based detectors are not effective against new or unknown malware. Another shortcoming is the size and maintenance of a signature database. Since a signature-based detector has to use
a separate signature for each malware variant, the database of signatures grows at an exponential rate. Moreover, frequent and recurrent updates of the malware signature database are imperative as new malware is churned out every day.

Figure 2.1. General Flow of Signature-based Malware Detection and Analysis

At the same time, we can see that in practice, a signature-based malware detection system uses the most expensive resource, namely the analyst, to analyse malware. In this situation, the classification of new malware by human analysis, whether through memorization, or looking up description libraries or searching sample collections is not an effective method as it is too time consuming and subjective [LM06]. The analyst has few tools to automatically classify a particular program into a specific family. To make matters worse, obfuscation technology is adopted by malware writers to evade being detected by anti-virus systems. As a consequence, developing an effective automatic malware detection and classification system would be significant for the anti-virus industry. In recent years, many researchers have turned their attention to the detection and classification of malware using many different approaches.
2.2 Related Work

In research of malware detection and classification, the two core problems to be resolved are:

1) Suitable Representation of Malware.


The representation of malware is heavily dependent on malware analysis and extraction approaches. Different analysis and extraction approaches focus on different aspects of malware and construct diverse feature sets. We must decide what kinds of information can be extracted from executables and how this information should be extracted, organized and used to represent the executables.

Choice of Optimal classification decision making is related to the classification algorithms and performance evaluation methods used. We need to decide which classification algorithms can be applied in our research and what is our generalized classification process, and at the same time decide how to evaluate the performance of our system.

All malware analysis and extraction approaches can basically be categorized into two types: (i) based on features drawn from an unpacked static version of the executable file without executing the analyzed executable files [Ghe05, KS06, SBN+10, PBKM07, XSCM04, XSML07, TBIV09, TBV08, WPZL09, YLCJ10, DB10, HYJ09] and (ii) based on dynamic features or behavioral features obtained during the execution of the executable files [CJK07, WSD08, AHSF09, KChK+09, ZXZ+10].
2.2.1 Static Feature Extraction

Since most of the commercial software and malware is distributed in the form of binary code, binary code analysis becomes the basis of static feature extraction. Traditional anti-virus detection and classification systems are based on static features extracted from executables by reverse-engineering [Eil05, FPM05, Eag08]. Static feature extraction based on binary code analysis is used to provide information about a program's content and structure which are elementary, and therefore a foundation of many applications, including binary modification, binary translation, binary matching, performance profiling, debugging, extraction of parameters for performance modeling, computer security and forensics [HM05].

As I mentioned previously, static feature extraction produces information about the content of the program, which includes code information, such as instructions, basic blocks, functions, modules, and structural information, like control flow and data flow. Much research is focused on this information from different perspectives. We consider a number of these below.

Gheorghescu [Ghe05] focuses on basic blocks of code in malware, which are defined as “a continuous sequence of instructions that contains no jumps or jump target”, and on average contains 12-14 bytes of data. These blocks are used to form a control flow graph. The author uses the string edit distance to calculate the distance between two basic blocks. The string edit distance between two basic blocks is defined as the number of bytes in which the blocks differ, which is also known as the edit distance. Similarity queries can be answered by computing the hash function for each basic block in the source sample and verifying whether the bit at the corresponding position in the target filter is set. The author presents two methods for approximate matching of programs. One is to compute the string edit distance, and another method is the inverted index which is commonly used in word search engines. As the author
mentions, these two methods have their drawbacks; edit distance is CPU-intensive and the inverted index is I/O bound. The bloom filters [Blo70] method was introduced because Bloom filters are efficient not only in query time but also in storage space because they are fixed in size. Bloom filters is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. Basic blocks are represented in a Bloom filter, and similarity queries can be answered by computing the hash function for each basic block in the source sample and verifying if the bit at the corresponding position in the target filter is set. Their results were presented on 4000 samples of Win32 malware. An important contribution of this paper is that the author demonstrates that it is possible to implement an automated real-time system to perform this analysis on a desktop machine.

Kapoor and Spurlock [KS06] argue that a binary code comparison of the malware itself is not satisfactory because it is error prone, can be easily affected by the injection of junk code and because code comparison algorithms are expensive with poor time complexity. They state that comparing malware on the basis of functionality is more effective because it is really the behavior of the code that determines what it is. Kapoor and Spurlock suppose that the more complex the function, the more likely it is to define the code behavior. Weightings are assigned to code depending on the complexity of the function. A function tree is then constructed based on the control flow graph of the system, and used to eliminate ‘uninteresting’ code. Then, they convert the tree description of a malware sample to a vector and compare vectors to determine the similarity of malware. The benefits of this are: control tree extraction and comparison, however a major drawback of this method is the intensive pre-processing which must be done in determining the weight assigned to each function.

In [SBN+10], the authors used weighted opcode sequence frequencies to calculate the cosine similarity between two PE executable files. These opcode sequences are based on static analysis and have two contributions. One is to assign a weight to each
opcode which computes the frequency with which the opcode appears in a collection of malware and benign software, then determines a ratio based on statistics. In this way, they mine the relevance of the opcode and also acquire a weight for each opcode. The second contribution is [SBN+10] proposes a method which relies on the opcode sequence frequency to compute similarity between two executable files. Their experiment was tested on a collection of malware downloaded from VxHeavens (http://vx.netlux.org) which comes from 6 malware families. In our opinion, code obfuscation is a big challenge for this method.

Several authors use sequences of system calls, API calls and function calls of malware to detect malicious behaviors. Peisert et al. [PBKM07] use sequences of function calls to represent the behavior of a program. Sathyanarayan et al. [SKB08] use static analysis to extract API calls from known malware then construct a signature for an entire class. The API calls of an unclassified sample of malware can be compared with the 'signature' API calls for a family to determine if the sample belongs to the family or not. The drawback is that obfuscation of API calls can affect the accuracy of results. In their paper Sathyanarayan et al. mention that they used IDA to extract API and they tested it on eight families with 126 malware in total. API Calls are also used by [XSCM04, XSML07] to compare polymorphic malware, with their analysis carried out directly on the PE (portable executable) code. API calling sequences are constructed for both the known virus and the suspicious code. In their method, they scan the whole section of CALL instructions for each code section of a PE file to obtain a set of strings, which stores the names of the called APIs. They then use Euclidean distance to perform a similarity measurement between the two sequences after a sequence realignment operation is performed.

Ye et al. [YLJW10] present a classifier using post-processing techniques of associative classification in malware detection which is based on their previous work they called Intelligent Malware Detection System (IMDS). Their method is based on
the static analysis of API execution calls. Their experiment was tested on a large collection of executables including 35,000 malware and 15,000 cleanware samples, and used various data mining techniques which achieved close to 88% accuracy.

[WPZL09] presents a virus detection technique based on identification of API call sequences under the windows environment. They first acquired API calls from malware files by static analysis of the procedures, and then set up the sequence of API calls. The authors choose Bayes algorithm as an approximate determinant of a virus because the Bayes algorithm is a method used to calculate the posterior probability according to prior probability. The machining learning method was applied during that procedure and the technique was a significant attempt to solve the win32 virus with low cure rate.

An intelligent instruction sequence-based malware categorization system is presented in [HYJ09]. It consists of three integrated modules: feature extractor, classification and signature generator. They used the IDA Pro disassembler to extract the function calls from the unpacked malware and a clustering method was used to classify. They tested their method on 2029 malware samples from 408 families and acquired close to 79% accuracy across their data set.

Further research of static analysis is presented in [DB10], which describes architecture for automated malware classification based on massive parallel processing of common code sequences found in static malware. In [DB10], only portions of this architecture have been implemented and the cost appears to be significant.

In our published paper [TBV08], we present a fast, simple and scalable method of classifying Trojans based only on the lengths of their functions. Our results indicate that function length may play a significant role in classifying malware, and combined with other features, may result in a fast, inexpensive and scalable method of malware classification. I will elaborate about function length in Chapter 4.
An effective and efficient malware classification technique based on string information is presented in our paper [TBIV09]. Using \textit{K-fold} cross validation on the unpacked malware and clean files, we achieved a classification accuracy of 97\%. Our results revealed that strings from library code (rather than malicious code itself) can be utilised to distinguish different malware families. In Chapter 5, further explanation of printable string information will be presented.

To make further progress, in [ITBV10] we combined the static features of function length and printable string information extracted by our static analysis methodologies. This test provides classification results better than those achieved by using either feature individually. We achieved an overall classification accuracy of over 98\%. Further description will be presented in Chapter 6.

\subsection*{2.2.2 Advantages and Disadvantages of Static Analysis}

In this section, I would like to elaborate on the advantages and disadvantages of static analysis and extraction. Static analysis and extraction of executable files provides information about the content and structure of a program, and therefore are the foundation of malware detection and classification. These have been well explored and widely adopted due to the following \textit{advantages}:

1) Low Level Time and Resource Consumption. During static analysis, we have no need to run the malware which is high in both time and resource Consumption. In static analysis, the time for disassembling is positively propagated with the size of the code, while the time for dynamic analysis is related to execution flow, which becomes even slower especially in the case of a loop with thousands and millions of iterations [Li04].

2) Global View. A huge advantage of static analysis is that it can analyze any
possible path of execution of an executable. This is in contrast to dynamic analysis, which can only analyze a single path of execution at a time. A static analysis has a good global view of the whole executable, covers the whole executable and can figure out the entire program logic of the executable without running it [Li04, Oos08].

3) Easily Accessible Form. In static analysis and extraction, we examine the disassembling code generated by reverse engineering software. The first step of reverse engineering software is usually to disassemble the binary code into corresponding assembler instructions, and then group these instructions in such a way that the content and structure information about the executable, like functions, basic blocks, control flow and data flow, is easily accessible [Oos08].

4) Stable and Repeatable. Compared with dynamic analysis, the disassembling code generated during static analysis is relatively stable and constant, which means is easy for us to apply and test new classification algorithms or theories. In addition, the repeatability of the disassembling procedure provides flexibility to static analysis and extraction.

5) Safety and Data Independent. Once the disassembling information of the original executable files is extracted and preserved, we do not need to operate the original files anymore. This means during static analysis, the opportunities of being affected by malicious code is reduced to zero.

While static analysis has its advantages, it also has its limitations:

1) Limitation of Software Reverse-engineering Techniques. Static analysis depends on software reverse engineering techniques, with the disassembling code of executables acted upon. The authors in [HCS09] mention that since most modern
malware programs are written in high-level programming languages, a minor modification in source code can lead to a significant change in binary code.

2) Susceptible to Inaccuracies Due to Obfuscation and Polymorphic Techniques. Code and data obfuscation poses considerable challenges to static analysis. More and more automated obfuscation tools implement techniques such as instruction reordering, equivalent instruction sequence substitution, and branch inversion. Malware authors can take advantage of these tools to easily generate new malware versions that are syntactically different from, but semantically equivalent to, the original version.

3) Content-based Analysis. Authors in [GAMP+08] argued that in the static analysis method, the representation of malware focuses primarily on content-based signatures, that is to say they represent the malware based on the structural information of a file, which is inherently susceptible to inaccuracies due to polymorphic and metamorphic techniques. This kind of analysis fails to detect inter-component/system interaction information which is quite important in malware analysis. At the same time, it is possible for malware authors to thwart content-based similarity calculations.

4) Conservative Approximation. The approximation is a standard static analysis technique, with this technique implemented with a few approximations which are always overly conservative [Sax07]. In addition, this approximation naturally involves a certain loss of precision [Vig07].

In [MKK07], the authors explore the limitation of the static analysis methodology from the point of view of obfuscation technology. In this paper, they introduce a code obfuscation schema which demonstrates that static analysis alone is not enough to either detect or classify malicious code. They propose that dynamic analysis is a
necessary complement to static techniques as it is significantly less vulnerable to code obfuscating transformations.

In [LLGR10], the authors point out that dynamic analysis of malware is often far more effective than static analysis. Monitoring the behavior of the binary during its execution enables it to collect a profile of the operations performed by the binary and offers potentially greater insight into the code itself if obfuscation is removed (e.g., the binary is unpacked) in the course of its running.

Increasingly, more researchers are now working on dynamic analysis techniques to improve the effectiveness and accuracy of malware detection and classification. In the next section, I will introduce some related dynamic analysis and extraction approaches.

2.2.3 Dynamic (run-time) Feature Extraction

In [CJK07], Christodorescu et al. argue that it is the behavior of malware that should be used to classify it. Viewing malware as a black box, they focus on its interaction with the operating system, thereby using system calls as the building blocks of their technique. They compare these with the system calls of non-malicious code in order to trim the resulting graph of dependencies between calls. In their method, behavioral information for each piece of malware has to be collected and a graph constructed for it. Their results are based on an analysis of 16 pieces of known malware.

The authors in [WSD08] use dynamic analysis technologies to classify malware by using a controller to manage execution, with the execution stopped after 10 seconds. Initially they calculated the similarity between two API call sequences by constructing a similarity matrix based on action codes (to our understanding action codes in this paper are actually the sequence of API calls). The relative frequency of each
function call was computed and the Hellinger distance was used to show how much information was contained in malware behavior to construct a second matrix. Finally, two phylogenetic trees were constructed using the similarity matrix and the Hellinger distance matrices separately. They tested this on a small set of 104 malware samples, and in my opinion, their algorithm has relatively high time and space complexities. In this paper, the authors do not mention the classification accuracy.

In [AHSF09], the authors open a new possibility in malware analysis and extraction by proposing a composite method which extracts statistical features from both spatial and temporal information available in run-time API calls. From the point of view of space, spatial features are generally statistical properties such as means, variances and entropies of address pointers and size parameters. From the perspective of time, the temporal feature is the nth order discrete time Markov chain [CT91] in which each state corresponds to a particular API call. They use 237 core API calls from six different functional categories and use a 10-fold cross validation procedure with five standard classification algorithms. The cost of their method is great because of its high computational complexity, while they archived good results with 96.3% classification accuracy.

A novel malware detection approach is proposed in [KChK+09], with the authors focusing on host-based malware detectors because these detectors have the advantage of observing the complete set of actions that a malware program performs and it is even possible to identify malicious code before it is executed. The authors first analyze a malware program in a controlled environment to build a model that characterizes its behavior. Such a model describes the information flow between the system calls essential to the malware’s mission and then extracts the program slices responsible for such information flow. During detection, they execute these slices to match models against the runtime behavior of an unknown program.
In [ZXZ+10], the authors propose an automated classification method based on behavioral analysis. They characterize malware behavioral profile in a trace report which contains the changed status caused by the executable and the event which is transferred from corresponding Win32 API calls and their parameters. They extract behavior unit strings as features which reflect behavioral patterns of different malware families. Then, these features of vector space serve as input to the support vector machine (SVM), with and string similarity and information gained used to reduce the dimension of feature space to improve system efficiency. They tested on 3996 malware samples and achieved an average classification accuracy of 83.3%.

In our published work [TIBV10], we provide our dynamic analysis and extraction methodology. We used an automated tool running in a virtual environment to extract API call features from executables and applied pattern recognition algorithms and statistical methods to differentiate between files. A more detailed explanation will be given in Chapter 7.

### 2.2.4 Advantages and Disadvantages of Dynamic Analysis

As with static analysis, dynamic analysis also has its advantages and disadvantages. Dynamic analysis outperforms static analysis due to the following characteristics.

*Advantages:*

1) Effectiveness and precision. Observation of the actual execution of a program to determine it is malicious is a lot easier than examining its binary code. Observation can reveal subtle malicious behaviors which are too complex to be identified using static analysis. Dynamic analysis is typically more precise than static analysis because it works with real values in the perfect light of run-time.
In addition, dynamic analysis is precise because no approximation or abstraction needs to be done and the analysis can examine the actual, exact run-time behavior of the executables [ME03].

2) Simplicity. Static analysis is the analysis of the source or the compiled code of an executable without executing it. It consists of analyzing code and extracting structures in the code at different levels of granularity. This is a very intensive process [Oos08]. As dynamic analysis only considers a single execution path, it is often much simpler than static analysis [Net04].

3) Runtime Behavioral Information. The advantage of dynamic analysis comes from the fact that malware executes its own designed-in functionality when it is started and when it is being executed. During dynamic analysis, the data collected, such as memory allocation, files written, registry read and written, and processes created, is more useful than the data collected during static analysis. The dynamic runtime information can be directly used in assessing the potential damage malware can cause which enables detection and classification of new threats. That is to say dynamic analysis can be used to probe the context information at run time since most values of register or memory can only be produced and watched on the fly [Li04].

4) No Unpacking. In dynamic analysis, we execute the malware in a controlled environment. During that procedure, malware automatically executes the unpacking code then runs its malicious code every time.

5) Robust to Code Obfuscation. Compared to static analysis, dynamic analysis is more effective in detecting obfuscated malware simply because the obfuscated code does not affect the final behavioral information collected during the execution.
In spite of the obvious advantages of dynamic detection methods in detecting modern malware, dynamic analysis detection has its own built-in drawbacks or *limitations*:

1) Limited View of Malware. It is time-consuming and impossible for us to examine all the possible execution paths and variable values during dynamic analysis, which means dynamic analysis provides a limited view of malware. Static analysis provides an overview of malware because the disassembling code contains all the possible execution paths and variable values.

2) Trace Dependence and Execution Time Period. The main limitation of any dynamic malware analysis approach is that it is trace-dependent [BCH+09]. The author states that in dynamic analysis, analysis results are only based on malware behavior during one (or more) specific execution runs. Unfortunately, some of malware’s behavior may be triggered only under specific conditions. In addition, the author provides three examples to illustrate its limitation. A simple example is a time-bomb, which is trigger-based and only exhibits its malicious behavior on a specific date. Another example is a bot that only performs malicious actions when it receives specific commands through a command and control channel. In addition, the time period in which malicious behaviors are collected in dynamic analysis is another limitation. It is possible that certain behaviors will not be observed within this period due to time-dependent or delayed activities [BOA+07].

3) Lack of Interactive Behavior Information. Since we run malware automatically without human interaction, interactive behaviors such as providing input or logging into specific websites, is not performed during dynamic analysis which limits the exhibition of further malicious behaviors [BCH+09].

4) Time and Resource Consumption. Dynamic analysis involves running malware
in a controlled environment for a specific time or until the execution is finished. It is a time-consuming task when we are faced with the challenge to analyse huge amounts of malware files released every day. In addition, running malware occupies a high level computer system and network resources [KK10].

5) Limitation of VM Environment and Detection Inaccuracy. The virtual machine environment in which malware is executed is relatively monotonous and steady compared with the real runtime environment, which also limits the exhibition of further malicious behaviors. Additionally, dynamic analysis detection about real functionality of the analyzed malware file can be inaccurate. [KK10].

2.2.5 Machine Learning based Classification Decision Making Mechanisms

Another core problem in malware detection and classification is the choice of mechanisms for classification decision making. This task involves the following aspects:

1) Selecting suitable classification algorithms.

2) Generalizing classification processes based on the feature sets obtained in malware analysis and extraction.

3) Evaluating the performance of system.

As I mentioned in Section 2.1, the main drawback of a signature-based detection system is that it cannot detect unknown malware. Machine learning is capable of generalizing on unknown data, and therefore can be a potential and promising approach for detecting malware. In order to detect unknown malware, more and more researchers are turning their attention in obtaining a form of generalization in malware detection and classification by using machine learning methodologies.
Machine Learning is defined by Ethem Alpaydin in [Alp04] as: “Machine Learning is programming computers to optimize a performance criterion using example data or past experience.” In [Nil96], the author points out that, like zoologists and psychologists who study learning in humans and animals, Artificial Intelligence researchers focus on learning in machines. The core idea of machine learning is generalization. In other words, machine learning is used to generalize beyond the known data provided during the training phrase to new data presented at the time of testing. Machine learning is a very open and practical field and it is broadly applied in many fields, including Expert System, Cognition Simulation, Network Information Service, Image Reorganization, Fault Diagnose, robotics, and machine translation.

[Sti10] points out that from a machine learning perspective, signature-based malware detection is based on a prediction model where no generalization exists, that is to say that no detection beyond the known malware can be performed. As we mentioned above, machine learning is capable of generalizing on unknown data, therefore, can be used in a malware detection system. In the current literature, many publications apply data mining and machine learning classification decision making mechanisms [AACKS04, MSF+08, SMEG09, CJK07, SEZS01, SXCM04, WDF+03, HJ06]. Machine learning algorithms used in this area include association classifiers, support vector machines, decision tree, random forest and Naïve Bayes. There have also been several initiatives in automatic malware categorization using clustering techniques [KM04].

The authors in [SEZS01] firstly introduce the idea of applying machine learning techniques in the detection of malware. They extract features from different aspects of malware, including the program header, string, byte sequence and four classifiers applied in their work, including the signature-based method, Ripper which is a rule-based learner, Naïve Bayes and Multi-Naïve Bayes. [SEZS01] found that machine learning methods are more accurate and more effective than signature-based methods.
In [AACKS04], the authors applied the Common N-Gram analysis (CNG) method which was successfully used in test classification analysis in the detection of malicious code. They adopted machine learning techniques based on byte n-gram analysis in the detection of malicious code. 65 distinct Windows executable files (25 malicious code, and 40 benign code) were tested and their method achieved 100% accuracy on training data, and 98% accuracy in 3-fold cross validation.

In [KM04], the authors applied machine learning techniques based on information retrieval and text classification in detecting unknown malware in the wild. After evaluating a variety of inductive classification methods, including Naïve Bayes, decision trees, support vector machines, and boosting, their results suggested that boosted decision trees outperformed other methods with an area under the ROC curve of 0.996.

The authors in [KM06] extended their previous work [KM04] by providing three contributions. To begin with they show how to use established data mining methods of text classification to detect and classify malicious executables. Secondly, they present empirical results from an extensive study of inductive machine learning methods for detecting and classifying malicious executables in the wild. Finally they show that their machine learning based methods achieved high detection rates, even on completely new, previously unseen malicious executables.

In [HJ06], the authors state that more general features should be utilized in malware detection because signatures are overfitted. They present a n-gram based data mining approach and evaluate their machine learning method by using 4 classifiers, including ID3 and J48 decision trees, Naïve Bayes and the SMO.

In [MSF+08], they employed four commonly used classification algorithms, including Artificial Neural Networks (ANN), Decision Trees, Naïve Bayes, and Support Vector Machines.
In [Sti10], they investigated the applicability of machine learning methods for detecting viruses in real infected DOS executable files when using the n-gram representation. Although, the author states that detecting viruses in real infected executable files with machine learning methods is nearly impossible in the n-gram representation. However, the author notices that learning algorithms for sequential data could be an effective approach, and another promising approach would be to learn the behaviors of malware by machine learning methods.

In [Kel11], the author investigated the application of a selection of machine learning techniques in malware detection. In [Kel11], the author states that we need a proactive approach not a reactive approach, which means malware should be identified before their signatures are known and before they have a chance to do damage. The preliminary results from their project supports the idea that AI techniques can indeed be applied to the detection of unseen malicious code with a feature set derived from Win32 API calls, and the results also provide evidence to the superiority of some techniques over others.

### 2.3 Our Proposed Method

As I mentioned in Section 2.2, the two core problems of malware detection and classification is suitable representation of malware and the choice of mechanism for optimal classification decision making. The first problem of representation of malware depends on the malware analysis and extraction approaches. The second problem of decision making mechanisms is related to the choice of classification algorithms, the generalization of classification processes and the evaluation of performance. My system aims to construct a robust, flexible and generalizable malware detection and classification system. In general, this system should have the following characteristics:
1) Malware analysis and extraction should be based on simple and effective feature sets.

2) These feature sets should be easily extracted, applied and combined.

3) These feature sets should present malware from different points of view in order to apprehend malware in an overall way as accurately as possible.

4) These features should be robust enough against the evolution of malware.

5) The malware classification process should be generalizable.

6) There should be an effective evaluation of performance.

Given the above factors and the two core problems that I mention above, we set up our malware detection and classification system. I provide a brief description of it in the following two sections.

2.3.1 Integrated Analysis and Extraction Approach

We start our system by using static analysis and extraction approaches. In static analysis, we analyze and extract simple and effective features from executable files, including Function Length Frequency (FLF), and Printable String Information (PSI). We cannot solely rely on one single approach, so therefore we introduce dynamic analysis and extraction approaches into our system. All of the features from both static and dynamic analysis are simple, can be easily extracted and are suitable to apply to both small and large data sets. We can obtain function length information by executing several database storage procedures and functions, and we can easily fetch printable string information from our table strings_window (see Section 3.4 in Chapter 3). These features can be easily combined and applied to a large data set and they also present malware from several different points of view. In addition, as we
will show from our experimental results in Chapters 4, 5, 6, 7, and 8, these features are effective and robust enough against the evolution of malware.

As all methods have their strengths and weaknesses, dynamic analysis does not aim to replace static analysis but does provide an additional layer of intelligence. Static and dynamic analysis should complement the merits of each other.

The author in [ME03] mentions that researchers need to develop new analytical methods that complement existing ones, and more importantly, researchers need to erase the boundaries between static and dynamic analysis and create unified analyses that can operate in either mode, or in a mode that blends the strengths of both approaches.

Our proposed method aims to build a robust system which integrates dynamic analysis and static analysis approaches. This allows the combination of their advantages and minimizes their imperfections. In our system, static analysis uses binary code analysis to examine the code which the malware is comprised and extract static features to capture the capabilities of the malware without actually executing it. Dynamic analysis provides a method for obtaining high-level, real-time behavior of an executable, which involves running the executable in a virtual machine environment. we then merge both static and dynamic features into a broader feature which is applied into our machine learning based classification decision making mechanism.

At the time of writing this thesis there was a lack of integrated malware detection and classification platforms which include complementary static and dynamic analysis in order to reach high and robust classification accuracy. Such an integrated platform is a primary contribution of this thesis.

In order to evaluate the robustness and scalability of our system, the malware executable files that we investigate stretch cross an 8-year span, from 2003 to 2010.
2.3.2 Machine Learning based Classification Decision Making Mechanisms

As I mentioned in Section 2.2.5, data mining and machine learning approaches are applied as a classification decision making mechanism in malware detection and classification in order to generalize the classification process, and therefore, to detect unknown malware.

In Section 3.6 of Chapter 3, I will provide a detailed description of machine learning and data mining techniques applied in our system. In this section, I will introduce four classification algorithms applied in our system: support vector machine (SVM), decision tree (DT), random forest (RF) and instance-based (IB1), along with boosting techniques. These algorithms represent the spectrum of major classification techniques available, based on differing approaches to the classification. Good classification accuracy obtained across all of these algorithms supports our claim of a robust methodology. In order to estimate the generalized accuracy, \(K\)-fold cross validation is applied because it is the most popular and widely used approach to measure how the results of a statistical analysis will be generalizable to an independent data set.

2.4 Hypotheses and Objective of System

Correctly classifying malware is an important research issue in a malware detection and classification system. As I discussed in Section 2.2, all the classification approaches can basically be categorized into two types: static methods and dynamic methods. In both static and dynamic methods, it is crucial to find key features that represent malware and are effective in malware detection and classification. Therefore, we propose the following hypotheses while setting up the malware detection and classification system.
• **Hypothesis 1**: It is possible to find static features which are effective in malware detection and classification. This hypothesis is verified by the FLF (Function Length Frequency) and FLP (Function Length Pattern) experiments presented in Chapter 4, and the PSI (Printable String Information) experiments presented in Chapter 5.

• **Hypothesis 2**: Combining several static features can produce a better detection and classification performance than any individual feature can produce. This hypothesis is verified by the combined static experiments presented in Chapter 6.

• **Hypothesis 3**: It is possible to find dynamic features which are effective in malware detection and classification. This hypothesis is verified by the dynamic experiments presented in Chapter 7.

• **Hypothesis 4**: Combining static and dynamic features can produce better detection and classification performance than any individual feature can produce. This hypothesis is verified by the integrated experiments presented in Chapter 8.

• **Hypothesis 5**: Good levels of malware detection and classification accuracy can be retained on malware collected over an extended period of time. In Chapter 8, this hypothesis is verified by the integrated experiments on two sets of malware collected from different time zones.

In reality, there is no such system that can achieve a 100% classification accuracy. Some researchers have achieved quite good results by testing different approaches on different data sets. For instance, Bailey et al. [BOA+07] achieved over 91% accuracy; Zhao, H., et al. [ZXZ+10] achieved an overall performance of approximately 83.3%
classification accuracy; Ye, Y. et. al. [YLJW10] achieved approximately 88% accuracy. For our malware detection and classification system, we would like to aim for 97% classification accuracy.

2.5 Summary

This chapter has presented a literature review related to my research. Based on the problem description of traditional methods used by the anti-virus industry, I proposed the two core problems of malware detection and classification. Following this, the current static and dynamic methods were presented and I analyzed and stated the merits and limitations of both static and dynamic approaches. I also proposed our integrated method and proposed our hypotheses along with the targeted classification accuracy of our system.
Chapter 3

Architecture of the System

3.1 Introduction

In this chapter, I explore our proposed malware detection classification system from the point view of system architecture. To make it easy to understand, I divide our system into different layers according to the flow of process with each layer having its own specific problems and corresponding solutions. First I outline these basic layers of our malware detection and classification system and then I explain them in detail section by section with a summary provided at the end of this chapter.

3.2 Overview of System

As I mentioned in Section 2.3 of Chapter 2, our aim is to construct a robust, flexible and scalable malware detection and classification system. It should be based on a set of simple and effective features which are easily extracted, applied and combined. While these features present malware from different points of view, at the same time these features should be robust enough against the evolution of malware. It also
should have a generalized malware classification process and an effective evaluation of performance. To set up such a system, we need to answer the following questions: where do we collect our test samples? How do we deal with these samples to make them suitable for our research? How do we store and maintain this data? We need to answer two core problems of a malware detection and classification system which I mentioned in Section 2.2 of Chapter 2. These are:

1) Suitable Representation of Malware.

2) Optimal Classification Decision Making Mechanisms.

Figure 3.1 provides a hierarchical structure of such a system. It is generalized into the following functional layers:

1) Data Collection and Data Preprocess Layer: In this layer, we decide where we collect test samples, including malicious software and benign software. We need to select representative samples which are then preprocessed to make sure they are well-formed and that they fit our research.

2) Data Storage Layer: We then need to choose a method to store this preprocessed data, taking data accessibility and manoeuvrability into consideration. At the same time, in the scenario of malware analysis, there is a potential risk of being affected by malware due to operational accidents. Therefore, in this layer, data security is also an important factor to consider in order to minimize this kind of risk.

3) Extraction and Representation Layer: Once we have the stored preprocessed data, we come to the core problem of malware detection and classification: what information should we extract from the executable files and how should we represent it in an abstract fashion based on the extraction.
Figure 3.1. Architecture of Our Malware Detection and Classification System
4) Classification Process Layer: On this layer, we need to select suitable classification algorithms. This classification process also needs to be generalized based on the feature sets obtained from the lower Extraction and Representation layer.

5) Performance Assessment Layer: The top layer is the statistical analysis of classification results and evaluation of performance.

In Figure 3.1, I outlined the architecture of our system. More detailed implementation information is presented in Figure 3.2. In this figure, I specify concrete implementation that is designed to meet the requirements mentioned above within each layer of our system.

Figure 3.2. Implementation of Our Malware Detection and Classification System
In the following section I explain the implementation of these layers in our system section by section.

3.3 Data Collection and Preprocess

In this section, I list the experimental dataset that we collected and explain the preprocess that we adopted in our research work.

3.3.1 Experimental Dataset

As I mentioned in Section 1.1.3 of Chapter 1, malware has become increasingly complicated and has evolved into more and more types. However some types, including Trojans, Viruses and Worms, are the most popular forms of malware and comprise the main security threats faced by host, network and application layers.

According to the analysis from Kaspersky Security Bulletin 2007 (http://www.securelist.com/en/analysis?pubid=204791987), in 2006 and 2007 the vast majority of malware were Trojans, Worms and Viruses. In their analysis report, TrojWare refers to Trojans and VirWare refers to Worms and Viruses.

From Figure 3.3 we can see that in 2006, the malware landscape was dominated by TroWare and Virware, which accounted for 89.45% and 6.11% respectively, and in 2007 the numbers were 91.73% and 5.64%. Figures 3.4 and 3.5 graphically show the distribution of malicious programs in 2006 and 2007.

Other anti-virus companies have also similar analyses. In 2009 and 2010, the Symantec company released the Global Internet Security Threat Report of trends for 2008 and 2009 respectively [Sym09, Sym10]. According to their report of trends for 2008, in the top 10 new malicious code families detected in 2008, six of them were
<table>
<thead>
<tr>
<th></th>
<th>2007</th>
<th>2006</th>
<th>% in 2007</th>
<th>% in 2006</th>
<th>Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrojWare</td>
<td>201958</td>
<td>91911</td>
<td>91.73%</td>
<td>89.45%</td>
<td>119.73%</td>
</tr>
<tr>
<td>VirWare</td>
<td>12416</td>
<td>6282</td>
<td>5.64%</td>
<td>6.11%</td>
<td>97.64%</td>
</tr>
<tr>
<td>MalWare</td>
<td>9798</td>
<td>4558</td>
<td>2.63%</td>
<td>4.44%</td>
<td>27.20%</td>
</tr>
<tr>
<td>Total</td>
<td>220172</td>
<td>102751</td>
<td>100%</td>
<td>100%</td>
<td>114.28%</td>
</tr>
</tbody>
</table>

Figure 3.3. Number of New Malicious Programs Detected by Kaspersky Lab in 2006 and 2007 [Gos08]

Figure 3.4. Distribution of Malicious Programs in 2006 [Gos08]

Figure 3.5. Distribution of Malicious Programs in 2007 [Gos08]
Trojans and three of them were Worms. According to their report of trends for 2009, of the top 10 new malicious code families detected in 2009, six were Trojans, three were Worms, and one was a Virus. In 2010 Trojans, Worms and Viruses still occupy the leading position. For example, the Quarterly Report [Pan10] from PandaLabs shows 77% of new malware identified by PandaLabs during the second quarter are Trojans, Viruses and Worms. Based on the analysis of these reports from the anti-virus community, the selection of our experimental dataset focuses on Trojans, Worms and Viruses.

Our project is supported by CA Technologies (www.ca.com). CA has a long history of research and development of anti-virus products. Its VET zoo contains huge amounts of malware collected by their research staff or provided by customers. All the malware in our system was collected from CA’s VET zoo and have been pre-classified using generally acceptable mechanical means. We chose three types of malware, including Trojans, Worms and Viruses. We also collected clean executables from Window platforms spanning Windows 98 to Windows XP, which we refer to as cleanware. Table 3.1 lists all the malware families and cleanware tested in our experiments.

From Table 3.1 of Chapter 3, we can see that our experimental data set includes Trojans, Viruses and Worms spanning from 2003 to 2010. I provide a detailed descriptions of each family in Appendix B. We divided them into two groups, the first collected between 2003 and 2008 including “Clagger”, “Robknot”, “Robzips”, “Alureon”, “Bambo”, “Boxed”, “Emerleox”, “Looked”, and “Agobot”, which we called them “Old Families”. The second group was collected between 2009 and 2010, including “Addclicker”, “Gamepass”, “Banker”, “Frethog”, “SillyAutorun”, “SillyDI”, “Vundo”, “Bancos”. We called these “New Families”.

The files that we collected were raw executable files, and were stored in the file
<table>
<thead>
<tr>
<th>Type</th>
<th>Family</th>
<th>Detection Date: starting ⇒ ending (YYYY-MM)</th>
<th>No. of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trojan</td>
<td>Bambo</td>
<td>2003-07⇒2006-01</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Boxed</td>
<td>2004-06⇒2007-09</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td>Alureon</td>
<td>2005-05⇒2007-11</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Robknot</td>
<td>2005-10⇒2007-08</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Clagger</td>
<td>2005-11⇒2007-01</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Robzips</td>
<td>2006-03⇒2007-08</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>SillyDl</td>
<td>2009-01⇒2010-08</td>
<td>439</td>
</tr>
<tr>
<td></td>
<td>Vundo</td>
<td>2009-01⇒2010-08</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>Gamepass</td>
<td>2009-01⇒2010-07</td>
<td>179</td>
</tr>
<tr>
<td></td>
<td>Bancos</td>
<td>2009-01⇒2010-07</td>
<td>446</td>
</tr>
<tr>
<td></td>
<td>adclicker</td>
<td>2009-01⇒2010-08</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Banker</td>
<td>2009-01⇒2010-06</td>
<td>47</td>
</tr>
<tr>
<td><strong>Subtotal of Trojan</strong></td>
<td></td>
<td></td>
<td><strong>1713</strong></td>
</tr>
<tr>
<td>Worm</td>
<td>Frethog</td>
<td>2009-01⇒2010-08</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>SillyAutorun</td>
<td>2009-01⇒2010-05</td>
<td>87</td>
</tr>
<tr>
<td><strong>Subtotal of Worm</strong></td>
<td></td>
<td></td>
<td><strong>261</strong></td>
</tr>
<tr>
<td>Virus</td>
<td>Agobot</td>
<td>2003-01⇒2006-04</td>
<td>283</td>
</tr>
<tr>
<td></td>
<td>Looked</td>
<td>2003-07⇒2006-09</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Emerleox</td>
<td>2006-11⇒2008-11</td>
<td>75</td>
</tr>
<tr>
<td><strong>Subtotal of Virus</strong></td>
<td></td>
<td></td>
<td><strong>424</strong></td>
</tr>
<tr>
<td><strong>Total of Malware</strong></td>
<td></td>
<td></td>
<td><strong>2398</strong></td>
</tr>
<tr>
<td>Cleanware</td>
<td></td>
<td></td>
<td><strong>541</strong></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>2939</strong></td>
</tr>
</tbody>
</table>

Table 3.1. Experimental Set of 2939 Files

system in the form of binary code. To make them suitable for our research, we needed to preprocess them. Because our system combines the static analysis method and dynamic analysis method, there were two ways of data preprocessing: static and dynamic.

### 3.3.2 Static Preprocess

In our static Preprocess, we unpacked the malware before passing them to IDA Pro, the reverse-engineering software.
Malware makers have always strived to evade detection from anti-virus software. Code obfuscation is one of the methods they use to achieve this. As time has passed, obfuscation methodologies have evolved from simple encryption to polymorphism, metamorphism and packing. Packing is becoming increasingly popular, with lots of packing tools available to malware makers, and more of them being created in short periods of time. At present, there are a few dozen different packing utilities available, most of which can be used by anyone with minimal computer skills. Each of these tools may have many variants or versions, for instance, UPX (http://upx.sourceforge.net/) has more than 20 known versions. There are several scrambler tools that can be used to create modified versions of UPX.

Although unpacking technology is beyond the scope of our research, we do use this technology in our project. The main idea of unpacking manually is to let the packed executable run until the unpacking procedure is finished. We then dump the unpacked executable from memory, with the dump hopefully occurring right after the executable has been unpacked completely. In the early stage of my research, I unpacked more than 200 packed malware files. There are many methods and tools that we can use to unpack a packed executable automatically, such as PEid (http://www.peid.info/) which can help us to find the common packer; UN-PACK (unpack.cjb.net), which is a set of tools for file analyzing and unpacking; .NET Generic Unpacker (http://www.ntcore.com/netunpack.php) which can dump .NET packed applications.

In our unpacking Pro-processing we used VMUnpacker 1.3. VMUnpacker 1.3 is free software from Sucop company (http://www.sucop.com), and is based on virtual machine technology, supporting 61 kinds of packers, including more than 300 versions. Figure 3.6 is the sectional drawing of VMUnpacker 1.3 running in a vir-
tual machine environment. After we obtained the unpacked executable files, we used reverse-engineering techniques to perform our static analysis.

![VMUnpacker 1.3](image)

**Figure 3.6. VMUnpacker 1.3**

### 3.3.2.2 Reverse Engineering Ida2sql

The main theory underlying static analysis is reverse-engineering. During static analysis, we capture the malware’s capabilities by examining the code from which the malware is comprised. We open the malware up with Reverse Engineering Software or disassemblers without actually executing it. Reverse-engineering is the reverse
process of compilation. To understand this, in Figure 3.7 I provide a brief description of these two processes: compilation and reverse-engineering.

In the compilation process, the high level source code text is initially broken into small pieces called tokens which are single atomic units of the language. These tokens are combined to form syntactic structures, typically represented by a parsing tree. The next stage is lexical, syntax and control flow analysis. During this stage, further syntactic information is extracted. The source code is then translated into assembly code by semantic analysis and an intermediate code generator, and finally, the generated code is optimized and the corresponding machine code is generated.

Reverse-engineering is the process of deriving a higher-level engineering description of its function in the form of source code or other specification from a compiled program. This can be used to provide insight into the execution plan and structure of a program and the algorithms used in the program. In the reverse-engineering process, the low level machine code of the executable is disassembled into assembly code, and then assembly code is decompiled into high level code afterwards.

Figure 3.7. Compilation and the Reverse-engineering Process

In the reverse-engineering analysis of malware, for the purpose of more specificity and accuracy, malware analysts always use the assembly code generated by the dis-
assembler. There are many reverse-engineering programs available, such as SoftICE [OF96], WinDBG [Rob99], IDA Pro [Eag08], OllyDBG [Yus04] etc. These disassemblers will allow you to safely view the executable code without executing it, and also allow you to write down the offsets of interesting breakpoints for further examination.

We chose IDA Pro as our main reverse-engineering analysis tool because IDA Pro is a Windows or Linux hosted, programmable, interactive, multi-processor disassembler and debugger that offers many features [Eag08] which are useful in this research. IDA Pro can identify known library routines which saves time and effort by focusing the analysis on other areas of the code, and its popular plug-ins make writing IDA scripts easier and allows collaborative reverse engineering. Furthermore, its built-in debugger can tackle obfuscated code that would defeat a stand-alone disassembler and can be customized for improved readability and usefulness. It is a popular reverse-engineering tool used by anti-virus companies, software development companies and military organizations.

As I mentioned above, the main theory and technology adopted in static analysis is reverse-engineering. In most cases, what we can obtain from reverse-engineering is the assembly code of executable files. Compared to high level programming language, assembly programming language is code intensive and is not easily understood. In order to make reverse-engineering easier, many people and organizations have developed applications which can be used as the intermediary agents to manage and exploit reverse-engineering information. Ida2sql is one of these applications.

Ida2sql was developed by a technical analyst from Zynamics GmbH (formerly SABRE Security GmbH) (http://www.zynamics.com/). It is the upgraded product of their previous tool named ida2reml which exports much of the contents of the IDA database to a XML-like file format. In order to store the disassembling information in a way that would be as least architecture dependent as pos-
sible and allow for fast querying, and at the same time, trying not to make it
too difficult to use directly through SQL, they developed ida2sql. Ida2sql is in-
cluded in BinNavi, which is a binary code reverse-engineering tool that was built
to assist vulnerability researchers who look for vulnerabilities in disassembled code
(http://www.zynamics.com/binnavi.html), but is also available as a stand-alone mod-
ule for anybody to use. Ida2sql is actually a Python module in charge of exporting
the disassembling information from IDA Pro into the SQL Schema.

Meanwhile, we found that ida2sql generates a set of tables for each module and
stores this information in MySQL. In order to make it more applicable to our research,
I upgraded this software to the following aspects:

1) Database Migration. I customized the software to support both MySQL and
the Microsoft SQL Server because the latter is more popular and more powerful
than the former.

2) Improve Database Schema. I altered the structure of the schema using a fixed
number of tables.

3) Export More Information. For the purpose of our research, I also customized
this software to export more information, such as printable string information,
and function complexity, etc.

Another improvement I made to the ida2sql-based data preprocess is the strength-
ening of the automation of our system through the development of a utility called “Al-
lenOne”. Ida2sql is implemented in IDAPython using python script and IDAPython
supports running Python scripts on start up from the command line. Such function-
ality is very useful when analyzing a set of binaries in batch mode. To make use of
this functionality, I developed the tool “AllEnOne”. Figure 3.8 is the interface of
this tool. With this tool we can manage the connection to our Ida2DBMS schema,
generate batch script exporting disassembling of large numbers of malware samples into Ida2DBMS at the same time, and execute sql script to fetch information from Ida2DBMS.

In Section 3.4, I provide more information on the customized database schema ida2DBMS.

![Figure 3.8. The Interface of AllEnOne](image)

### 3.3.3 Dynamic Preprocess

In the dynamic analysis, we execute each file under a controlled environment which is based on Virtual Machine Technology. We developed a trace tool named “HookMe” to monitor and trace the real execution of each file.
3.3.3.1 Virtual Machine Environment

Figure 3.9 illustrates our dynamic analysis preprocess. We can see that the dynamic analysis preprocess is based on Virtual Machine Technology. To begin with, executable files location information is fetched from our ida2DBMS schema and is then passed to our VMrun-based Dynamic Analysis Script which controls the executions of the executables in the VMware environment. VMrun is the command-line utility of VMware [VMw09]. The Dynamic Analysis Script, written based on VMrun, automates the controlling of the executions in a batch mode. In Section 7.3.1 of Chapter 7, I provide a detailed description of this implementation.

To set up our virtual machine environment, we install and configure VMware Server 2.0.2. We then created a new virtual machine by installing Window XP Professional as a guest operation system and also disabled networking. Before the start of dynamic analysis, a snapshot of the virtual machine was taken and we needed to revert to the snapshot for every execution. In this way, we were assured that the virtual machine was rehabilitated every time.

After the execution, we obtained a log file which reflected the behaviors of the malware in terms of API function calls. In the next section, I discuss related technology we applied in the trace tool HookMe.

3.3.3.2 Trace tool

We developed a trace tool which we called "HookMe" to monitor and trace the execution of malware. It is built around Microsoft technology called Detours [HB99] which is designed to intercept Win32 functions by re-writing target function images and performs API hooking of functions that are imported into an executable. Basically, Detours is a library used to easily instrument and extend the operating
system and application functionality. Detours technology provides three important functionalities:

1) Intercepts arbitrary Win32 binary functions on x86 machines.

2) Edits the import tables of binary files.

3) Attaches arbitrary data segments to binary files.

HookMe is implemented based on the first functionality. There are three concepts we need to understand in this technology: Target Functions, Trampoline Functions and Detour Functions.

- **Target Functions.** Target Functions are the functions that we want to intercept or hook and are usually windows API functions.

- **Trampoline Functions.** These are actually copies of Target Functions, with instructions from the Target Functions preserved in Trampoline Functions. Trampoline Functions consist of the instructions removed from the Target Functions.
and an unconditional branch to the remainder of the Target Functions. In this way, Trampoline Functions keep the semantics of Target Functions.

- **Detour Functions.** A Detours function replaces the first few instructions of the target function with an unconditional jump to the user-provided Detour Function. Detour Functions are designed by the user to replace or extend the Target Functions.

In [HB99], the authors describe the logic flow for control of function invocation with and without interception which I present in Figure 3.10. From this figure, we can see that the detour function replaces the target function, but it can invoke its functionality at any point through the trampoline.

HookMe uses Detours to hook selected functions, and focuses on configuring Detours to specify what to collect and directs output to a log file. HookMe may automatically trace various events which happen during the execution of the malware.
Figure 3.11. Trace Tool HookMe

It actually monitors the state changes in the system and generates a log file which reflects the behaviors of the malware in terms of API function calls. Figure 3.11 illustrates the implementation of Detours technology in HookMe which we applied in our dynamic analysis. HookMe runs the monitored executable file and rewrites the in-process binary image of the related API calls invoked in the executable file. In this way, HookMe intercepts the windows API calls, gets the information of API calls and records this information in the log file.
3.4 Data Storage

In this section, I will present the implementation of the Data Storage Layer in our system. As I mentioned in Section 2.3 of Chapter 2, our proposed method aims to build a robust system which integrates both dynamic analysis and static analysis approaches. There are two ways for data storage in our system corresponding to these two different methods of analysis.

3.4.1 Static Data Storage

As I mentioned in 3.3.2.2, I customized and improved ida2sql to meet our research requirements. We chose database management system DBMS (http://www.microsoft.com/sqlserver/2005/en/us/default.aspx) as our data storage because DBMS has the following benefits which facilitate our work:

- **Integrated Management Interface.** All the information we need in our static analysis is stored in DBMS, so that we achieve a united environment where we can easily access the binary information. At the same time, we can expand our data collection simply by running a command or running a plugin in IDA. In addition, we can utilize the functions of DBMS backup and restore. Furthermore, our customized static analysis software supports analysis of a set of executable files in an automatic and batch mode. In practice, we can schedule a large amount of analysis when the server is idle.

- **Standard Form for Binaries.** Data structure of binaries in a database is consistent with the structure of the program, including functions, basic blocks of a function, instructions and operators.
• Fetch Data in a Simple and Effective Way. Using sql script, we can query or fetch information, such as a specific instruction, all the instructions of a function, or all the basic blocks of a function, etc. Writing sql script is much easier and less error-prone than writing plugins for IDA.

• Faster to Test New Algorithms/Theories. Because all the information is stored in the database, when a new algorithm or theory is developed, we need to extract specific data from the database. This is much faster than testing the new algorithm or theory on the original files.

• Data Independent. Once the binary information of the original files is analysed and put into the database, we no longer need the original files. That means the opportunities for being infected by malicious code is reduced to zero.

As I mentioned above, there are many benefits when we use DBMS to store disassembling information. For instance, we can fetch information we are interested in just by executing a sql script. To explain it more clearly, I present several examples in Table 3.2, 3.3, 3.4, 3.5, and 3.6 with each table displaying sql script. Following is the execution result of that sql script.

We can obtain the basic information from all the files of a specific family, shown in Table 3.2. After executing the sql script, we obtain a list of basic information for all the files from the “Clagger” family. We can obtain the main function information of files as explained in Table 3.3. After executing the script, the main function information of some executable files assigned in the script can be fetched. In this example, we obtain main function information by assigning ID values to corresponding files. We can also fetch the list of instructions of a Basic Block from a File. The example in Table 3.4 explains how to obtain the list of instructions of a basic block by assigning the module ID value and basic block ID value in sql script. Table 3.5
select * from modules where family='clagger'.

<table>
<thead>
<tr>
<th>ID</th>
<th>FILENAME</th>
<th>ENTRYP</th>
<th>TYPE</th>
<th>FAMILY</th>
<th>VARIANT</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>connect_unpacked.exe</td>
<td>4203506</td>
<td>trojan</td>
<td>Clagger</td>
<td>A</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>fiks_unpacked.exe</td>
<td>4203506</td>
<td>trojan</td>
<td>Clagger</td>
<td>A</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>gzwazd2h_unpacked.exe</td>
<td>4206594</td>
<td>trojan</td>
<td>Clagger</td>
<td>A</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>2853315_unpacked.exe</td>
<td>4220146</td>
<td>trojan</td>
<td>Clagger</td>
<td>AB</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>3044352_unpacked.exe</td>
<td>4205752</td>
<td>trojan</td>
<td>Clagger</td>
<td>AD</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>ID0220712_unpacked.exe</td>
<td>4201170</td>
<td>trojan</td>
<td>Clagger</td>
<td>AE</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>Ebay-Rechnung.pdf_unpacked.exe</td>
<td>4228109</td>
<td>trojan</td>
<td>Clagger</td>
<td>AG</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>DD269901_unpacked.exe</td>
<td>4201106</td>
<td>trojan</td>
<td>Clagger</td>
<td>AH</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>TT-022-421-683_unpacked.exe</td>
<td>4204066</td>
<td>trojan</td>
<td>Clagger</td>
<td>AI</td>
<td>...</td>
</tr>
<tr>
<td>11</td>
<td>2727905_unpacked.exe</td>
<td>4202706</td>
<td>trojan</td>
<td>Clagger</td>
<td>AJ</td>
<td>...</td>
</tr>
<tr>
<td>12</td>
<td>2803317_unpacked.exe</td>
<td>4202354</td>
<td>trojan</td>
<td>Clagger</td>
<td>AK</td>
<td>...</td>
</tr>
<tr>
<td>13</td>
<td>web_unpacked.exe</td>
<td>4203698</td>
<td>trojan</td>
<td>Clagger</td>
<td>AL</td>
<td>...</td>
</tr>
<tr>
<td>14</td>
<td>photoalbum_unpacked.exe</td>
<td>4215506</td>
<td>trojan</td>
<td>Clagger</td>
<td>AM</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>xpfaease_unpacked.exe</td>
<td>4206274</td>
<td>trojan</td>
<td>Clagger</td>
<td>AN</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.2. Fetch Basic Information of Executable Files from a Specific Family

shows how to fetch an instruction located at the specific address and Table 3.6 shows how to fetch all the printable string information from a file.

From these examples, it is evident that extracting information from executable files becomes much easier and flexible after we exported the executables into our ida2DBMS schema.

After the static data preprocessing, we exported the disassembling information into a database schema which we called ida2DBMS. Figure 3.12 illustrates this schema. In this schema, there are 19 tables. The main tables are Modules, Functions, Basic Blocks, Instructions, Strings, window, fun_len, callgraph, and control_flow_graph. Every entry in the Modules table represents an executable file, and it describes attributes of an executable file, including storage location in the file system, family and variant information, entry point, platform, MD5 values and import time etc. Each entry in the Functions table contains information describing a specific func-
select modules.module_id, modules.name, modules.entry_point,
functions.name fname, functions.address, functions.end_address
from functions, modules
where functions.module_id = modules.module_id
and modules.entry_point between functions.address and functions.end_address
and modules.module_id in (2,3,4).

<table>
<thead>
<tr>
<th>ID</th>
<th>FILENAME</th>
<th>ENTRYP</th>
<th>FNAME</th>
<th>FSTARTADDR</th>
<th>FENDADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>connect_unpacked.exe</td>
<td>4203506</td>
<td>start</td>
<td>4203506</td>
<td>4203793</td>
</tr>
<tr>
<td>3</td>
<td>fiks_unpacked.exe</td>
<td>4203506</td>
<td>start</td>
<td>4203506</td>
<td>4203793</td>
</tr>
<tr>
<td>4</td>
<td>gzwazd2h_unpacked.exe</td>
<td>4206594</td>
<td>start</td>
<td>4206594</td>
<td>4206881</td>
</tr>
</tbody>
</table>

Table 3.3. Fetch Main Function Information of Some Assigned Executable Files

select instructions.module_id, instructions.basic_block_id,
instructions.address, operand_id, position,
instructions.mnemonic, str from operand_tuples, instructions, operand_strings
where operand_tuples.module_id = instructions.module_id
and instructions.module_id = operand_strings.module_id
and instructions.module_id = 2
and operand_tuples.address = instructions.address
and operand_tuples.operand_id = operand_strings.operand_string_id
and instructions.basic_block_id = 146.

<table>
<thead>
<tr>
<th>ID</th>
<th>BASIC_BLOCK_ID</th>
<th>ADDR</th>
<th>OPER_ID</th>
<th>POS</th>
<th>MNEMONIC</th>
<th>STR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>146</td>
<td>4203340</td>
<td>14234</td>
<td>0</td>
<td>inc</td>
<td>edi</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203341</td>
<td>14235</td>
<td>0</td>
<td>mov</td>
<td>edx</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203341</td>
<td>14236</td>
<td>1</td>
<td>mov</td>
<td>[esp+8+arg_0]</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203345</td>
<td>14237</td>
<td>0</td>
<td>neg</td>
<td>eax</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203347</td>
<td>14238</td>
<td>0</td>
<td>neg</td>
<td>edx</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203349</td>
<td>14239</td>
<td>0</td>
<td>sbb</td>
<td>eax</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203349</td>
<td>14240</td>
<td>1</td>
<td>sbb</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203352</td>
<td>14241</td>
<td>0</td>
<td>mov</td>
<td>[esp+8+arg_4]</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203352</td>
<td>14242</td>
<td>1</td>
<td>mov</td>
<td>eax</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203356</td>
<td>14243</td>
<td>0</td>
<td>mov</td>
<td>[esp+8+arg_0]</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>4203356</td>
<td>14244</td>
<td>1</td>
<td>mov</td>
<td>edx</td>
</tr>
</tbody>
</table>

Table 3.4. Fetch the List of Instructions of a Specific Basic Block from a File
Figure 3.12. Idb2DBMS Schema
select instructions.module_id, instructions.address, operand_id, position, instructions.mnemonic, str from operand_tuples, instructions, operand_strings 
where operand_tuples.module_id = instructions.module_id 
and instructions.module_id = operand_strings.module_id 
and instructions.module_id = 2 
and operand_tuples.address = instructions.address 
and operand_tuples.operand_id = operand_strings.operand_string_id 
and instructions.address = 4203341.

<table>
<thead>
<tr>
<th>ID</th>
<th>ADDR</th>
<th>OPER_ID</th>
<th>POS</th>
<th>MNEMONIC</th>
<th>STR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4203341</td>
<td>14235</td>
<td>0</td>
<td>mov</td>
<td>edx</td>
</tr>
<tr>
<td>2</td>
<td>4203341</td>
<td>14236</td>
<td>1</td>
<td>mov [esp+8+arg_0]</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5. Fetch the Specific Instruction

tion, such as the start and end address of the function, or the type of function etc. A function is composed of many basic blocks which are described in Basic Blocks table. ID value and address information of the basic block are also provided in this table. Similarly, each basic block is composed of many instructions which are described in the Instruction table. String_window table represents printable string information in each executable file. The Callgraph table contains the call information among functions and the control_flow_graph table contains information of the logic relationship among basic blocks in a function. The fun_len table describes length information of functions. In Appendix D, I list all the tables and provide detailed information for each table.

3.4.2 Dynamic Data Storage

In this sub-section I provide a detailed description of data storage in our dynamic analysis. As I mentioned in Section 3.3.3, we ran each executable file in the VM environment, traced that execution and wrote down the intercepted windows API calls in a log file. Figure 3.13 gives an example of such log files. In the log file, each line records a windows API call, including Timestamps, the name of API call and corresponding parameters.
select * from strings_window where module_id =2.

Output of Execution Result

<table>
<thead>
<tr>
<th>SID</th>
<th>ID</th>
<th>SECCNAME</th>
<th>ADDR</th>
<th>STRLEN</th>
<th>STRTYPE</th>
<th>STR</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>2</td>
<td>.newIID</td>
<td>4223096</td>
<td>13</td>
<td>0</td>
<td>ADVAPI32.dll</td>
</tr>
<tr>
<td>138</td>
<td>2</td>
<td>.newIID</td>
<td>4223112</td>
<td>14</td>
<td>0</td>
<td>egSetValueExA</td>
</tr>
<tr>
<td>139</td>
<td>2</td>
<td>.newIID</td>
<td>4223128</td>
<td>12</td>
<td>0</td>
<td>RegCloseKey</td>
</tr>
<tr>
<td>140</td>
<td>2</td>
<td>.newIID</td>
<td>4223144</td>
<td>15</td>
<td>0</td>
<td>enProcessToken</td>
</tr>
<tr>
<td>141</td>
<td>2</td>
<td>.newIID</td>
<td>4223161</td>
<td>22</td>
<td>0</td>
<td>LookupPrivilegeValueA</td>
</tr>
<tr>
<td>142</td>
<td>2</td>
<td>.newIID</td>
<td>4223185</td>
<td>22</td>
<td>0</td>
<td>AdjustTokenPrivileges</td>
</tr>
<tr>
<td>143</td>
<td>2</td>
<td>.newIID</td>
<td>4223209</td>
<td>16</td>
<td>0</td>
<td>RegCreateKeyExA</td>
</tr>
<tr>
<td>144</td>
<td>2</td>
<td>.newIID</td>
<td>4223225</td>
<td>13</td>
<td>0</td>
<td>kernel32.dll</td>
</tr>
<tr>
<td>145</td>
<td>2</td>
<td>.newIID</td>
<td>4223240</td>
<td>13</td>
<td>0</td>
<td>GetLastError</td>
</tr>
<tr>
<td>146</td>
<td>2</td>
<td>.newIID</td>
<td>4223256</td>
<td>17</td>
<td>0</td>
<td>etCurrentProcess</td>
</tr>
<tr>
<td>147</td>
<td>2</td>
<td>.newIID</td>
<td>4223276</td>
<td>7</td>
<td>0</td>
<td>inExec</td>
</tr>
<tr>
<td>148</td>
<td>2</td>
<td>.newIID</td>
<td>4223285</td>
<td>8</td>
<td>0</td>
<td>lstrlen</td>
</tr>
<tr>
<td>149</td>
<td>2</td>
<td>.newIID</td>
<td>4223296</td>
<td>20</td>
<td>0</td>
<td>etWindowsDirectoryA</td>
</tr>
<tr>
<td>150</td>
<td>2</td>
<td>.newIID</td>
<td>4223320</td>
<td>18</td>
<td>0</td>
<td>tSystemDirectoryA</td>
</tr>
<tr>
<td>151</td>
<td>2</td>
<td>.newIID</td>
<td>4223340</td>
<td>17</td>
<td>0</td>
<td>GetModuleHandleA</td>
</tr>
<tr>
<td>152</td>
<td>2</td>
<td>.newIID</td>
<td>4223360</td>
<td>15</td>
<td>0</td>
<td>etStartupInfoA</td>
</tr>
<tr>
<td>153</td>
<td>2</td>
<td>.newIID</td>
<td>4223377</td>
<td>25</td>
<td>0</td>
<td>CreateToolhelp32Snapshot</td>
</tr>
<tr>
<td>154</td>
<td>2</td>
<td>.newIID</td>
<td>4223404</td>
<td>15</td>
<td>0</td>
<td>Process32First</td>
</tr>
<tr>
<td>155</td>
<td>2</td>
<td>.newIID</td>
<td>4223421</td>
<td>12</td>
<td>0</td>
<td>CloseHandle</td>
</tr>
<tr>
<td>156</td>
<td>2</td>
<td>.newIID</td>
<td>4223436</td>
<td>16</td>
<td>0</td>
<td>eternminateProcess</td>
</tr>
<tr>
<td>157</td>
<td>2</td>
<td>.newIID</td>
<td>4223456</td>
<td>17</td>
<td>0</td>
<td>tExitCodeProcess</td>
</tr>
<tr>
<td>158</td>
<td>2</td>
<td>.newIID</td>
<td>4223476</td>
<td>18</td>
<td>0</td>
<td>etModuleFileNameA</td>
</tr>
<tr>
<td>159</td>
<td>2</td>
<td>.newIID</td>
<td>4223496</td>
<td>14</td>
<td>0</td>
<td>Process32Next</td>
</tr>
<tr>
<td>160</td>
<td>2</td>
<td>.newIID</td>
<td>4223512</td>
<td>12</td>
<td>0</td>
<td>OpenProcess</td>
</tr>
<tr>
<td>161</td>
<td>2</td>
<td>.newIID</td>
<td>4223524</td>
<td>10</td>
<td>0</td>
<td>MFC42.DLL</td>
</tr>
<tr>
<td>162</td>
<td>2</td>
<td>.newIID</td>
<td>4223534</td>
<td>11</td>
<td>0</td>
<td>MSVCRT.dll</td>
</tr>
</tbody>
</table>

Table 3.6. Fetch All the Printable String Information from a File
After we obtained our ida2DBMS schema and log files of dynamic analysis, we no longer needed the original executable files. Next I will present the Extraction and Representation Layer of our system.

### 3.5 Extraction and Representation

As I mentioned above, once we finished the data preprocessing, the exporting of static disassembling information and recording of dynamic execution information, we no longer needed the original executable files. We did however extract static features from DBMS and extract dynamic features from log files. In this section, I will explain them individually.
3.5.1 Static Features Extraction and Representation

Static features are extracted from DBMS. In Section 3.4.1, I introduce our ida2DBMS schema, which holds lots of information we can extract from it.

As I mentioned in Section 2.3 of Chapter 2, the feature sets in our system would be simple, effective and easily extracted, expanded and combined. At the same time, these feature sets would present malware from different points of view and would be robust enough against the evolution of malware. Keeping these in mind, we set up our system by extracting function distance, function length and printable string information.

3.5.1.1 Functions

In the context of IDA Pro, function refers to an independent piece of code identified as such by IDA. In IDA and other disassembling software, segment code analyzed into smaller functional units are generally called “functions”. A function is essentially a basic unit of code which reflects the intention of the code. I use two aspects of function; one is the distance between functions and the another is function length related information.

The first experiment was called Function Distance Experiment. In this experiment, I extracted all the binary code of each function from an executable file, and then use three distance algorithms to calculate the similarity between any two functions from different executables. Please refer to Appendix A for further detailed information on this experiment.

Function length is defined by IDA and equates to be the number of bytes in the function. We define the function length pattern vector of an executable as a vector representing the length of all the functions in this executable. Each component in
the vector represents the length of a function and all the components are arranged in ascending order. The raw function length pattern vectors are of different sizes and scale, therefore are not directly comparable. In order to apply function length pattern vectors into our system, we used two different approaches to creating vectors of a standardized size:

1) Function Length Frequency. In this experiment, we counted the frequency of functions of different lengths.

2) Function Length Pattern. In this experiment, we standardized the function length vectors to be of the same size and scale so patterns could be compared.

In Chapter 4, I provide a detailed description of these two experiments.

3.5.1.2 Printable String Information

Another feature that we investigated was printable string information. A string is a consecutive sequence of printable characters, with this definition often augmented to specify a minimum length and a special character set. The strings utility is designed especially to extract string content from files. IDA Pro has a built-in equivalent of the strings utility, which we used to extract the strings from executables. In Appendix D, I provide detailed information of printable strings in our table named strings_windows, and I present this experiment in Chapter 5.

Following this, we did another experiment based on combined static feature extraction. This experiment is described in Chapter 6.
3.5.2 Dynamic Features Extraction and Representation

After we achieved the dynamic execution log files for executables, we used the dynamic analysis approach to extract and represent features. The purpose of our feature extraction approach is to separate out all the API call strings along with their parameters which occurred in the log files. We treated the API calls and their parameters as separate strings. In Chapter 7, I provide a detailed description of our dynamic analysis experiment.

3.6 Classification Process

As I mentioned in Section 2.2 of Chapter 2, another core problem of the malware detection and classification system is the Classification Decision Making Mechanisms, which is the generalization of the classification process. As discussed in Section 2.2.5 of Chapter 2, the core idea of Machine Learning is generalization, and Machine Learning is capable of generalizing unknown data, and therefore can be a potential and promising approach for detecting malware. We adopted Machine Learning and Data Mining techniques in our system.

The Machine Learning method is divided into two stages: one is to construct the classifier, which is called the training or learning phase, with the data used in this phase called training or learning data. The second stage is to evaluate, which is called validation. The data on which we evaluate the classifier is called test data. By applying Machine Learning techniques into our classification system, we need to answer the following questions:

1) Feature Extraction: In the context of Machine Learning, feature extraction refers to what we select as input to the computer. This issue was discussed in the Feature Extraction and Representation Layer of our system in Section 3.5.
2) Setup Training Set: The machine learning process is a process of learning. We adopt a supervised learning algorithm, which means there is labeled data in their training set.

3) Setup classifier models by using Classification Algorithms: The selection of classification algorithms.


3.6.1 **K-fold Cross Validation**

We adopt the **K-fold** cross validation method in our training and testing phases. **K-fold** cross validation [Koh95a] is a class of model evaluation methods in statistical analysis. Its purpose is to indicate how well the learner will perform when it is asked to make new predications for data it has not already seen. The idea is not to use the entire data set when training a learner as some of the data is taken away before training begins. When training is finished, this data can be used to test the performance of the learned model in a way that estimates how accurately a predictive model will perform in practice. It involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, multiple rounds of cross validation are performed using different partitions, and the validation results are averaged over the rounds.

**K-fold** cross validation is the most popular type of cross validation. In **K-fold** cross validation, each **K** sub-samples is used exactly once as the validation data. The **K** results from the folds can then be averaged or otherwise combined to produce a single estimation. The advantage of this method over repeated random sub-sampling
is that all observations are used for both training and validation, and each observation is used exactly once for validation.

We adopted the supervised Machine Learning approach in our automatic malware detection and classification system. The learning procedure includes training procedure and validation procedure. In order to evaluate the classification model generated during the training procedure, we chose \textit{K-fold} cross validation to do the validation test. When we used the \textit{K-fold} cross validation method, we needed to decide how many folds to employ. When \( K \) is small, this means the model has a small amount of data to learn from. When \( K \) is large this means the model has a much better chance of learning all the relevant information in the training set. While no theoretical result supports any particular choice of the value of \( K \), in practice, people always chose \( K = 2, 5, 10 \) [ARR05]. We used \textit{5-fold} cross validation in the experiments in Chapters 4, 5 and 6. In order to increase the chance of learning all the relevant information in the training set, we adjusted to \textit{10-fold} cross validation in our later dynamic experiment in Chapters 7 and 8. That is to say in our experiments, we randomly partitioned the test data into 5 or 10 subsets of approximately equal size, with one subset then used as the test set, and the other 4 or 9 subsets combined as the training set. We then used the training set to calibrate the test and validate the effectiveness against the test set. This procedure is normally repeated 5 or 10 times.

3.6.2 Selection of Classification Algorithms

WEKA (Waikato Environment for Knowledge Analysis) is a popular machine learning workbench with a development life of nearly two decades and widely accepted in both academia and business [BFH+10, HFH+09]. WEKA contains more than 100 classification methods which are divided into five main categories. These are
the Bayesian methods, lazy methods, rule-based methods, tree learners and function-
based learners. In our system, we chose five kinds of Machine Learning classification
algorithms, one from each of the above five categories. There were NB (Naïve
Bayesian classifiers), IB1 (Instance-Based Learning), DT (Decision Tree), RF (Random
Forest), and SVM (Support Vector Machine). All of these classifiers are basically
learning methods that adopt sets of rules. In addition we use AdaBoost, which is an
ensemble of classifiers in conjunction with these learning algorithms to improve their
performance. I elaborate on these algorithms in the next section.

3.6.2.1 Naïve Bayesian (NB)

Naïve Bayesian classifiers are derived from Bayesian Decision Theory [DHS00].
This is a widely used classification method due to its manipulation capabilities and
associated probabilities according to the user’s classification decisions and empirical
performance. In Bayesian classifiers, each class is represented with a single probabilis-
tic summary. The assumption is that all the attributes of the class are conditionally
independent and so the presence (or absence) of a particular attribute of a class is
unrelated to the presence (or absence) of any other attribute. That is to say that the
Naïve Bayesian considers that all of the attributes independently contribute to the
probability summary of classification. Because of the precise nature of the probability
model, Naïve Bayesian classifiers can be efficiently trained in a supervised learning
setting.

In [LIT92], the authors presented an average-case analysis of the Bayesian classifier
and gave experimental evidence for the utility of Bayesian classifiers. They concluded
that in spite of their naïve design and apparently over-simplified assumptions with
a comparative lack of sophistication, Bayesian classifiers deserve increased attention
in both theoretical and experimental studies. Naïve Bayesian classifiers have worked
quite well in many complex applications, such as Bayesian spam filtering which makes use of a Naïve Bayesian classifier to identify spam e-mails [SDHH98].

In [SEZS01], the authors designed a data-mining framework to train multiple classifiers on a set of malware and cleanware to detect new malware. They adopted a static analysis method and extract three kinds of static features, including system resource information obtained from the program header, printable string information extracted by the GNU strings program [PO93] and byte sequences. Then they applied RIPPER [Coh96], Naïve Bayesian and Multi-Naïve Bayesian to train on a set of malware of 3265 malware and 1001 cleanware to set up the classifiers and evaluate using the 5-fold cross validation. They showed that the Naïve Bayes algorithm using strings as features performed the best out of the learning algorithms and better than the signature method in terms of false positive rate and overall accuracy. Since the Naïve Bayes and Multi-Naïve Bayes methods are probabilistic, they mentioned that these algorithms could tell if an executable had similar probabilities to be classified as malware or cleanware. In this case, they could set up an option in the network filter to send a copy of the executable for further analysis by anti-virus analysts.

In [WDF+03], the authors proposed an automatic heuristic method to detect unknown computer viruses based on data mining techniques including Decision Tree and Naïve Bayesian classifiers. Their results showed that both perform well in terms of detection rate and accuracy.

The authors in [CGT07] compared four data mining classification algorithms over seven feature selection methods based on the byte sequence frequencies extracted from executables. Although the results from their experiments showed that SVM is superior in terms of prediction accuracy, training time, and aversion to overfitting, the Naïve Bayesian classifier still performed well.

In [MER08], the authors examined whether a machine learning classifier trained
on behavioral data collected from a certain computer system was capable of correctly classifying the behaviors of a computer with other configurations. They chose four machine learning algorithms, including Naïve Bayes, Decision Trees, Naïve Bayes, Bayesian Networks and Artificial Neural Networks. From the results of their eight experiments, they demonstrated that current machine learning techniques are capable of detecting and classifying worms solely by monitoring host activity.

We chose Naïve Bayesian classifiers as one of our classification algorithms in the static analysis methods presented in Chapters 5 and 6. They consider all of the attributes to independently contribute to the probability summary of classification which may not be the case for our data. This limitation may affect the accuracy of classification and results in Chapters 5 and 6 prove this hypothesis. In those experiments, Naïve Bayes gave the weakest results which led us to exclude Naïve Bayes in later experiments.

### 3.6.2.2 Instance-Based Learning IB1

IB1 is the simplest IBL (Instance-Based Learning) algorithm which is commonly known as KNN (k-nearest neighbours algorithm) and is proposed by Aha et al. [AKA91]. Instance-based learning or memory-based learning generates classification predictions using only specific instances instead of performing explicit generalizations, and compares new problem instances with instances seen in training which have been stored in memory. It extends the nearest neighbour algorithm by significantly reducing the storage requirement. Because it constructs a hypotheses directly from the training instances in memory, the IBL algorithm is very simple and effective. It is popularly used in many applications.

IBL algorithms have several advantages. One of them is simplicity, which allowed us to use a detailed analysis to guide our research motivation and aims. IBL
algorithms have a relatively relaxed concept bias and low updating costs [AKA91]. Another advantage is that IBL supports relatively robust learning algorithms. It can tolerate noise and irrelevant attributes and can represent both probabilistic and overlapping concepts. [AK89].

In [LM06], the authors proposed a behavior-based automated classification method based on distance measure and machine learning. They used the string edit distance as the distance measure, and then performed the nearest neighbor classification. They tested this in two separate experiments, using 461 samples of 3 families and 760 samples of 11 families and ran 10-fold cross validation on the above two datasets. They found that even though the string edit distance measure was costly, IBL classification performed quite well and they emphasized the importance of developing an automated classification process that applies classifiers with innate learning ability on near lossless knowledge representation.

The authors in [Ghe05] evaluated IBL with three different distance measures from the perspective of run time, storage space, and classification accuracy. Their tests demonstrated that it is possible to build an automated real-time system that can answer malware evolutionary relationship queries and run on an average desktop machine.

In [Weh07], the authors adopt IBL in classifying worms by using a normalized compression distance (NCD) measure. To assign the family of an unknown worm, they compute the NCD between the worm and all available binary worms. The family best matched is the family of worms which is closest to the unknown worm in terms of NCD. The authors in [See08] propose using machine learning approaches to learn malware intention and specific functionality properties. Their method is based on recorded execution traces in a sandbox in virtual environments, with focus on the research of spam and botnets. In their preliminary experiments, they tested many
different learning algorithms, including Naïve Bayes, Logistic Regression, IBL, and the rule learner JRip, however they discussed that IB1 offered the best performance.

In [AACKS04], the authors applied the CNG (Common N-Gram) method [KPCT03] based on byte n-gram analysis in the detection of malware. They chose the most frequent n-gram with their normalized frequencies to represent a class profile and used the KNN classification algorithm. Their results tested 65 distinct windows executable files (25 malware and 40 cleanware) and achieved 100% accuracy on training data and 98% accuracy in 3-fold cross validation.

Based on the analysis from works of other researchers, along with the advantages of the IBL algorithms mentioned above, we choose IB1, which is the simplest IBL algorithm as one of our malware classification algorithms.

3.6.2.3 Decision Table (DT)

A decision table (also known as a logic table) or a decision tree (also known as a decision diagram) describes the conditions associated with particular actions or decisions, along with constraints on the associated behavior. A decision table shows rules as rows and conditions as columns, with an entry in each cell for each action that should be taken by the person performing the behavior. A decision tree shows decision points as diamonds (as on a flowchart) and actions as boxes [Got02]. Decision tables have been advocated as a programming tool for displaying logical relationships and programmers frequently use decision tables because they translate directly into programmable logic [BD72, Lan78].

The Decision Table classification algorithm builds a simple decision table majority classifier, and summarizes the dataset or sample space with a decision table which contains the same number of attributes as the original dataset or sample space. Following this a new data item is assigned a class by finding the line in the decision
table that matches the non-class values of the data item. This was proposed by Ron Kohavi [Koh95b].

The Decision Table classifier is a spreadsheet-like classifier which is easy for humans to understand. In [KS98], the authors improved this algorithm by using entropy-based attribute selection to replace their previous methods based on a forward selection of attributes using the wrapper model. In their experiment they showed that prediction accuracy of a decision table majority classifier is comparable to that of widely used induction algorithms, such as C4.5.

In our system, we chose Decision Table as our rule-based classification algorithm.

### 3.6.2.4 Random Forest (RF)

Random Forest is a classifier consisting of a collection of tree-structured classifiers [Bre01]. It is an ensemble classifier that consists of many decision trees and outputs the class that is the mode of the class’s output by individual trees. In this instance, “mode” is a statistical term and is the value that occurs most frequently in a data set or a probability distribution. Random Forest actually grows many classification trees and in order to classify a new object from an input vector, the input vector puts down each of the trees in the forest, and each tree gives a classification. That is, the tree “votes” for that class, and then the forest chooses the classification with the most votes over all the trees in the forest.

Each decision tree in the Random Forest is a classifier in the form of a tree structure, where each node is either a leaf node or a decision node. A leaf node indicates the value of the classification and a decision node specifies the test to be carried out on a single attribute-value, with one branch and sub-tree for each possible outcome of the test. In general terms, the purpose of the analysis via tree-building algorithms is to determine a set of if-then logical conditions that permit accurate
prediction or classification of cases. A decision tree can be used to classify a sample by starting at the root of the tree and moving through until reaching a leaf node which provides the classification value. A random classifier uses a number of such decision trees in order to improve the classification rate.

In the decision tree algorithm, the main topics are selection of attributes on a decision node and method for splitting the tree. There are several popular algorithms used to generate a decision tree, such as ID3 [Qui86], C4.5 [Qui93], CART [BFSO84] and CHAID [Kas80]. ID3 and C4.5 are based on information theory, which C4.5 is an extension of Quinlan’s earlier ID3 algorithm and C4.5 builds decision trees from a set of training data in the same way as ID3 by using the concept of information entropy. It chooses attributes for which entropy is minimum or information gain is maximum in order to most effectively split its set of samples into subsets. CART (Classification and regression trees) was first introduced by Breiman et al. in 1984. Decision trees are formed by a collection of rules based on values of certain attributes in the modeling data set and these rules are selected based on how well splits based on attribute values can differentiate observations based on the dependent attribute. CART’s methodology is technically known as binary recursive partitioning. Binary refers to the fact that all decisions involve a parent node that is broken into 2 child nodes. Recursive refers to the fact that once a rule is selected and splits a node into two, the same logic is applied to each “child” node and in so doing, a decision tree with expanding branches is created. CHAID (Chi-squared Automatic Interaction Detector) is one of the oldest tree classification methods originally proposed by Kass (1980). According to Ripley (1996), the CHAID algorithm is a descendant of THAID, developed by Morgan and Messenger. CHAID builds non-binary trees on a relatively simple algorithm that is particularly well suited for the analysis of larger datasets and is a recursive partitioning method.

The author of the dissertation [Sid08] presented a data mining framework to detect
malicious programs. The author applied an array of classification models also used in our experiments, including Logistic Regression, Neural Network, Decision Tree, Support Vector Machines and Random Forest. Random forest outperforms all the classifiers and dimension reduction methods in their experiments in terms of overall accuracy, false positive rate and area under the ROC curve.

In [TKH+08], the authors introduced two classification models, one based on Support Vector Machine (SVM) and the other on Random Forest (RF) to detect malicious email servers. Their experimental results showed that both classifiers are effective, with RF slightly more accurate at the cost of time and space.

The authors in [HGN+10] proposed an Intelligent Hybrid Spam-Filtering Framework (IHSFF) to detect spam by analyzing only email headers. They applied various machine learning algorithms over five features extracted from the email header to set up the framework. Their experimental results showed that RF algorithm performed well in terms of accuracy, recall and precision.

Due to its excellent performance in classification tasks and in wider applications in information security, we chose RF as one of our classification algorithms.

### 3.6.2.5 Support Vector Machine (SVM)

SVM (Support Vector Machine) is a powerful, state-of-the-art algorithm with strong theoretical foundations based on Vapnik’s theory [Vap99]. It has a strong data regularization property and can easily handle high dimensional feature spaces. SVM is based on the Structural Risk Minimization (SRM) principle in order to find an optimal hyperplane by maximizing the margins that can guarantee the lowest true error due to increasing the generalization capabilities [DHS00]. The high classification accuracy of SVM is due to the fact that the learning algorithm in SVM not only minimizes the training error but also maximizes the generalization [Bur98]. The SVM method
we used in our system is Sequential Minimal Optimization which is a fast method to solve huge quadratic programming problems and is widely used to speed up the training of the Support Vector Machines.

SVM was firstly applied to detect malicious code in [ZYH+06] where they tested a total number of 632 samples, including 423 cleanware and 209 malware. Their experimental results show that the SVM based method can be effectively used to discriminate normal and abnormal API function call traces.

The authors in [SKF08a, SKF08b] used traffic patterns collected from real and simulated worm traffic. They compared six different classifiers based on these features including three bio-inspired classifiers, two statistical malware detectors and SVM. Their results showed that the best classification results are obtained with the SVM classifier. The authors also point out that SVM is an ideal candidate to act as a benchmark in their comparative study because of its high classification accuracy even though the SVM has high algorithmic complexity and extensive memory requirements.

In [YCW+09], the authors developed an interpretable string based malware detection system (SBMDS) using a SVM ensemble with bagging or Bootstrap aggregating [Bre96]. In their experiments, the SVM method also works well in classification.

In [ZZL+09], the authors proposed a system calls tracing system based on the full virtualization via Intel-VT technology and SVM. They use SVM to process system call sequences extracted from 1226 malicious and 587 benign executables to detect unknown executables. Their SVM based experiment shows that the proposed method can detect malware with strong resilience and high accuracy.

The authors in [WWE09] also applied SVM in their malware analysis based on both process-related information and the executed system calls.

In [ALVW10], the authors applied SVM to train the classifier and derived an
optimum n-gram model for efficiently detecting both known and unknown malware. They provided a preliminary analysis by using SVM and n-gram. They tested that on a dataset of 242 malware and 72 cleanware, and with their experiments providing a promising accuracy of 96.5%.

The authors in [SKR+10] also mentioned that data mining has been a recent focus of malware detection research. They present a data mining approach by applying SVM classification model based on the analysis of the frequency of occurrence of each Windows API.

We introduce SVM as one of our selected classification algorithms based on the above survey.

3.6.2.6 AdaBoost

An ensemble of classifiers is a set of classifiers whose individual decisions are combined in some way, typically by weighted or unweighted voting, to classify new examples [Dit97]. Because uncorrelated errors made by the individual classifiers can be removed by voting, ensembles can improve classification accuracy in supervised learning [Dit97].

The Boosting method is a well established ensemble method for improving the performance of any particular classification algorithm. Boosting is a general and provably effective framework for constructing a highly accurate classification rule by combining many rough and moderately accurate hypotheses (called weak classifiers) into a strong one. The weak classifiers are trained sequentially and, conceptually, each is trained mostly on the examples which were more difficult to classify by the preceding weak classifiers.

The concept of boosting an algorithm was initially presented in [Sch90] by Schapire, who provided the first provably polynomial-time boosting algorithm. On-
going research introduced a new generation of Boosting methods called AdaBoost (Adaptive Boosting) [Fre95, FS95, FS97]. This method was a much more efficient boosting algorithm and solved many of the practical difficulties of earlier boosting algorithms.

We also introduced the AdaBoost method into our system in order to improve the classification performance.

In sum, the above analysis of the literature on classification algorithms has influenced our choice of five algorithms. There are NB (Naïve Bayesian classifiers), IB1 (Instance-Based Learning), DT (Decision Tree), RF (Random Forest), and SVM (Support Vector Machine ). In addition, we also applied AdaBoost to each of these to determine if indeed it would improve our results.

3.7 Performance Assessment

The upper layer of our system is Performance Assessment. In this layer, I discuss the methods to assess the system performance based on the classification results. As I mentioned above, our classification process system is based on Machine Learning and Data Mining techniques. During the classification process, we can always arbitrarily label one class as a positive and the other one as a negative class. The experimental set is composed of \( P \) positive and \( N \) negative samples for each family.

As I discussed in Section 3.6, Machine Learning method is divided into two stages: the training or learning phase, and the evaluation phase. The classifier built during the training phase assigns a class to each sample from the test set, with some assignments possibly correct but some of the assignments possibly wrong. To assess the classification results, we count the number of true positive (TP), true negative (TN), false positive (FP) (actually negative, but classified as positive) and false negative
Figure 3.14. Performance of Classification

(FN) (actually positive, but classified as negative) samples [Vuk]. Figure 3.14 shows the relationship between them. In this figure, a dot represents a positive class and the star represents a negative class. The test set is composed of two sets $\mathcal{P}$ and $\mathcal{N}$. $\mathcal{P}$ has $P$ positive samples and $\mathcal{N}$ has $N$ negative samples. All the data within the ellipse are the samples being labelled as class one and all the data outside the ellipse are the samples being labelled as another class. We can see that samples in $\mathcal{P}$ are composed of $TP$ samples from the original positive set and $FP$ samples from the original negative set. Samples in $\mathcal{N}$ are composed of $FN$ samples from the original positive set and $TN$ samples from the original negative set. Some measures are introduced by the following formulas to assess the performance of the classifier. These are $TPrate$, $FPrate$, $FNrate$, $Precision$ and $Accuracy$:

$$\begin{align*}
TPrate &= TP/P \\
FPrate &= FP/N \\
FNrate &= FN/P \\
Precision &= TP/(TP + FP) \\
Accuracy &= (TP + TN)/(P + N)
\end{align*}$$

(3.7.1)
The aim of our system was to achieve a high classification accuracy. If accuracy equals 100%, this meant we classified samples completely and correctly. As I discussed in Section 2.4 of Chapter 2, our target was 97% classification accuracy for our malware detection and classification system.

3.8 Robustness of System

Robustness is one of the key factors of an automated malware detection and classification system. Malware trouble makers use obfuscation technologies to evade detection, but they always focus on some specific features of malware. For example, they could confuse the disassembler at the instruction level by junk instruction insertion, transform of unconditional jumps and call instructions to the respective branch functions [LDS03], however these tricks cannot conceal some other features, such as string information extracted from the executables. Thus, we expect our system to be robust because we combined not only static features, FLF (Function Length Feature) and PSI (Printable String Information), but also dynamic API call sequence features. These features complement each other.

In practice, anti-virus detection faces a big challenge with huge amounts of malware released daily. Anti-virus detection systems need to provide a timely response before the malware can cause any damage to the system. Under such circumstances, knowing whether a file is malicious or clean is more urgent than knowing the specific family from which the malware emanates. A robust malware detection and classification system should provide a timely response by differentiating malware from cleanware. In our system, we not only classified malware into different families, but also differentiate malware from cleanware.

In the research literature on malware detection and classification, it is suggested
that malware becomes immune to older anti-virus detection methods over time. To test the integrated system to see if it was robust to changes in malware evolution, we introduced more recent malware into our test dataset and divided the dataset into two groups according to their age (measured when the executable file was first collected) and then tested these two groups in our integrated experiments. We expected our integrated system to be robust through the inclusion both old and new malware families.

### 3.9 Summary

In this chapter, I described our automatic malware detection and classification system from the point view of the hierarchical architecture of a system. Our system was divided into five layers, which are the Data Collection and Data Preprocess Layer, Data Storage Layer, Extraction and Representation Layer, Classification Process Layer and the Performance Assessment Layer. I explained the problem and corresponding solution for each layer.

Firstly I described our experimental dataset and due to our system using a combination of static and dynamic methodologies, I then presented both static and dynamic methods in the Data Collection and data Preprocess Layer, Data Storage Layer and Extraction and Representation Layer. In the Classification Process Layer and Performance Assessment Layer, I described the application of machine learning and data mining methods in our system. Finally, I gave a brief description of the expected robustness of our system.
Chapter 4

Function Length Features based Methodology

As I explained in Section 3.5 of Chapter 3, there were two methods of features extraction and representation in our system; one was static and another was dynamic. This chapter focuses on the function based static features extraction and representation.

4.1 Introduction

Malware is essentially a piece of program code. Making malware should follow the principle of computer programming techniques. A computer program actually is a list of instructions that tell a computer how to accomplish a set of calculations or operations [VRH04]. A program is often composed of self-contained software routines which perform a certain task as defined by a programmer. Such self-contained software routines or *functional units* have many different names, such as procedure, method or function. In order to get an overall acknowledge of a piece of program code,
first we need to understand each functional unit. Functional unit is the foundation of analyzing a program.

An IDA function is simply an independent piece of code identified as such by IDA and is not necessarily software which performs a certain task as defined by a programmer, while IDA function is essentially the functional unit of disassembling code, corresponding to the functional unit of high level program code. Therefore IDA function is the foundation when we analyze disassembling code of the malware. We then start our research work from IDA functions.

In this chapter, I present my method of classifying malware based only on the lengths of their functions. Two aspects of function length are investigated: one is the length of the function as measured by the number of bytes of code in it; the other is the frequency with which function lengths occur within any particular executable file of malware. These values are easy to obtain as output from IDA for any unpacked input. Our results indicate that both function features are significant in identifying the family to which a piece of malware belongs; the frequency values are slightly more significant than the function lengths with respect to accuracy while the reverse is true with respect to the rate of true positives.

In Section 4.2, I summarize the relevant literature in this area. In Section 4.3, I detail our analytical approach and data preparation and in Section 4.4 I describe the experimental set-up for our two function length based. Sections 4.5 and 4.6 present the two tests individually. In Section 4.8 I analyse and compare the results of the two tests and extended one and summarize in Section 4.9.
4.2 Related Work

As I described in Chapter 2, to set up an automatic malware detection and classification system, many researchers used static features drawn from unpacked executables without executing them [Ghe05, KS06, SBN+10, PBKM07, XSCM04, XSML07, TBIV09, TBV08, WPZL09, YLCJ10, DB10, HYJ09].

Gheorghescu [Ghe05] used basic blocks of code in the malware to form a control flow graph. Kapoor and Spurlock [KS06] compared vectors converted from a function tree which is constructed based on the control flow graph of the system to determine similarity of malware. Peisert et al. [PBKM07] used sequences of function calls to represent program behaviour. Sathyanarayan et al. [SKB08] used static analysis to extract API calls from known malware in order to construct a signature for an entire class. API Calls were also used by [XSCM04, XSML07, WPZL09, YLJW10] to compare polymorphic malware. Kai Huang et al. [HYJ09] developed an instruction sequence-based malware categorization system. Igor Santos et al. [SBN+10], used weighted opcode sequence frequencies to calculate the cosine similarity between two PE executable files.

In the following section, I present our approach to the classification problem based on the static features extracted from function length. On the positive side, our methods are based on simple features extracted from binaries, including function length frequency and function length pattern. On the negative side, using function size and frequency appears to give a correct classification in only about 80% of cases and so these features must be used with others for a better determination.
4.3 Data Preparation

This approach is based on the prototype system proposed in Chapter 3. In the Data Collection and Data Preprocess Layer of our system I first unpack the malware executables using the free software program named VMUnpacker v1.3 and in a few cases I unpack the packed executables using manual processes. Then disassemble them using IDA, export the disassembling analysis to our database schema ida2DBMS in the Data Storage Layer. Our architecture allows us to effectively extract large amounts of disassembling information and obtain a wide range of features of a malware in a swift and simple way in the Extraction and Representation Layer of our system. Our aim is to use features that are simple and inexpensive to extract, so we start from IDA function length information. To easily understand this process, I first explain the disassembling process of IDA Pro and IDA function.

4.3.1 IDA Disassembling Process

Figure 4.1 is a brief description of IDA disassembling process.

![IDA Disassembling Process Diagram]

Three main components of IDA Pro, including loader module, disassembling engine and processor module, play the important role in the disassembling process. IDA
loader modules behave much as operating system loaders behave\(^1\). There are three types of loader modules in IDA Pro:

- **Window PE loader.** Used to load Window PE files.
- **MS DOS EXE loader.** Used to load MS DOS EXE files.
- **Binary File.** Default for loading files that are not recognized by reading header structure in the disk image of the analyzed file.

Once you have chosen a file to analyze, the selected loader module starts to load the file from disk, parse any file-header information that it may recognize, create various program sections containing either code or data as specified in the file header, identify specific entry points into the code. In such a way that the selected loader module determines a virtual memory layout for the disk image of the analyzed file and then the selected loader module returns control to IDA.

Once the loading has finished, the disassembling engine takes over and begins to pass address from the virtual memory layout to the selected processor module one by one. In most cases, IDA chooses the proper processor module based on the information that it reads from the executable file’s headers or you can assign a proper processor type before IDA starts to analyze the file.

It takes processor module two passes to finish generating the assembly code for the analyzed file. In the first pass, the process module determines the type and length of instruction located at that address and the locations at which execution can continue from that address. In such a way, IDA detects all the instructions in the file. In the second pass, processor module generates assembly code for each instruction at each address.

\(^1\)For more information about operating system loader, please refer to [Pie94, Pie02]
4.3.2 IDA function

Functions are identified by IDA through the analysis of addresses that are targets of call instructions. IDA performs a detailed analysis of the behaviour of the stack pointer register to understand the construction of the function’s stack frame. Stack frames are blocks of memory, allocated within a program’s runtime stack and dedicated to a specific invocation of a function [Eag08]. Based on the analysis of the layout of function’s stack frame, IDA identifies each function.

We know that in high-level programming, programmers typically group executable statements into function units, including procedures, subroutines or methods, which perform a certain task as defined by the programmers. So function units are always the basis when people do high-level program analysis.

An IDA function is simply an independent piece of code identified as such by IDA and it is not necessarily a function unit. However IDA is a well-defined mechanism that maps high-level programming constructs into their messy, assembly code equivalents. So we assume that IDA functions have quite similar characteristics as functional unit, then we choose IDA functions as the basis of our static analysis and extraction.

4.3.3 Extract Function Length Information

The first step that we need to do is to fetch data information of each function for an executable file from our ida2DBMS schema. Figure 4.2 describes five tables involved in the formation of the function data information. These five tables are Instructions, BasicBlocks, Functions, Modules and FunctionLength. From this figure we can see that an IDA function is composed of many basic blocks; and each basic block is composed of instructions. All the instructions and all the basic blocks that
belong to a function are traversed and put them together to form the data of the function.

Figure 4.2. IDA Function Data

To get function length information, I program the following four database functions or stored procedures to fetch function length information from the database:

- **GetBasicblockData.** A database function used to fetch all the instruction information to form data information for a specific basic block.

- **GetFunData.** A database function used to fetch all the basic block data information to form data information for a specific function.

- **GetFileData.** A database storage procedure used to extract function length information for all the functions of an executable file and store this information in database.
Figure 4.3. Related Database Programs

- *GetFamilyData*. A database storage procedure used to extract function length information for all the executable files of an assigned family.

Figure 4.3 illustrates the calling relationship between these four database programs.

The core program of these four programs is “GetFunData”. To extract function length information, we need to generate data information for each function by execute “GetFunData”. Figure 4.4 describes the process of generating of data information for each function.

For each function in the disassembling module of a specific executable file, first we need to get all the basic blocks belonging to that function; and then identify and
Figure 4.4. Function Fetch
fetch all the instructions of each basic block by using the value of basic_block_id. Then combine these instructions to form a hex format string for each basic block, in the same way, combine all the basic blocks belonging to the function to form a hex format string representing the function.

In Appendix D I provide the detailed description of our Ida2DB schema and these four database programs mentioned above.

4.4 Experimental Set-up

4.4.1 Motivation

In the initial stages we extracted function length information from our ida2DBMS database schema. For each malware executable file we constructed a list containing the length (in bytes) for all the functions. We then sorted the list from the shortest length function to the longest, and graphed it. We call this the function length pattern. Figure 4.5 illustrates three samples from the Robzips family and Figure 4.6 illustrates three samples from the Robknot family.

![Figure 4.5. Function Length Pattern Samples from Robzips Family](image)

With malware executables from within the same malware family, we noticed that
although the number of functions and their lengths varied, the shape of the function length pattern looked similar. The executables from different families have different patterns. This motivated us to investigate whether function length contains statistically significant information for classifying malware.

The unpacking preprocess mentioned in Section 3.3.2.1 of Chapter 3 may not produce the original binary. In addition, when IDA disassembles unpacked malware, it identifies functions according to its own auto-analysis procedure. The functions finally extracted may be different from those returned by the malware programmer. Although it is difficult to be precise in regard to exactly what is meant by a function in the context of our experiments, we are nevertheless using a reliable and repeatable process.

In our experiments, function length is defined to be the number of bytes in the function as defined by IDA. The function length pattern vectors are the raw input given to our experiments. An example function length vector, taken from the Beovens family, is (24, 38, 46, 52, 118, 122, 124, 140, 204, 650, 694, 1380). (All vectors and sets referred to in this paper are ordered.) Each component in this vector represents the length of a function in the example. There are 12 functions in the sample, and the
function lengths are 24, 38, 46, \ldots, 1380 respectively; the maximum function length is 1380.

### 4.4.2 Test Dataset

When we first do function length based experiment, we use 721 files from 7 families of Trojans. Table 4.1 lists the families in this experiment. In this experiment our aim is to investigate whether function length contain statistically important information for classifying malware. So we start with a relatively small test dataset collected over a 4-year span which is from 2003 to 2007.

<table>
<thead>
<tr>
<th>Family</th>
<th>Detection Date: starting ⇒ ending (YYYY-MM)</th>
<th>No. of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bambo</td>
<td>2003-07⇒2006-01</td>
<td>41</td>
</tr>
<tr>
<td>Boxed</td>
<td>2004-06⇒2007-09</td>
<td>263</td>
</tr>
<tr>
<td>Alureon</td>
<td>2005-05⇒2007-11</td>
<td>43</td>
</tr>
<tr>
<td>Beovens</td>
<td>2005-03⇒2007-06</td>
<td>144</td>
</tr>
<tr>
<td>Robknot</td>
<td>2005-10⇒2007-08</td>
<td>101</td>
</tr>
<tr>
<td>Clagger</td>
<td>2005-11⇒2007-01</td>
<td>47</td>
</tr>
<tr>
<td>Robzips</td>
<td>2006-03⇒2007-08</td>
<td>82</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2003-07⇒2007-11</strong></td>
<td><strong>721</strong></td>
</tr>
</tbody>
</table>

Table 4.1. Experimental Set of 721 Malware Files

### 4.4.3 Overview of Experimental Process

Figure 4.7 provides an overview of our function length based experiments.

The raw function length vectors are of different sizes so are not directly comparable. We try two different approaches to creating vectors of standardized size. The first is to count the frequency of functions of different lengths (described in Section 4.5), the other is to standardize the function length vectors to be of the same size and
scale so that the patterns could be compared (described in Section 4.6). Figure 4.7 is the overview of these two experiments.

In order to determine whether function length information can be used in classification, we choose, in each test and for each family, a target vector, which we call a ‘centroid’ and determine how close each sample is to this centroid. For a good choice of the centroid, we expect samples in the family to be close in a carefully defined statistical way, and we expect samples not in the family to be far.

We use \textit{K-fold} cross validation in each test. For each family we randomly partition the vectors into 5 subsets of approximately equal size. We use one subset as the test set, and combine the other 4 subsets as the training set. We use the training set to calibrate the test and validate the effectiveness of the centroids against the test set. This is repeated 5 times, so that each vector is used as a test sample.

Our classification uses an adaptation of the technique described by [SKB08]. For
each training set we calculate a centroid vector. We use statistical methods to determine whether a test vector is sufficiently close to the centroid vector to be classified as belonging to that family.

4.5 Function Length Frequency Test

We first introduce some standard notation which is referred to throughout this section and the next. Let $P = \{P_1, P_2, \ldots, P_N\}$ represent a general population of $N$ function length vectors. $P^* = \{P_1^*, P_2^*, \ldots, P_N^*\}$ represents the set of all the standardized vectors. We use $F$ for a set of $n$ vectors from a specific family having $n$ samples. For any particular function vector $P_k$ with $m_k$ elements, we write $P_k = (p_{k1}, p_{k2}, \ldots, p_{km_k})$ and refer to $m_k$ as the size of the function length vector.

In both the tests of Sections 4.5 and 4.6, we use the 5-fold cross validation method discussed in Section 3.6.1 of Chapter 3, applying it five times. In both cases, $T = \{T_1, T_2, \ldots, T_r\} \subset F$ represents a training set chosen from the family $F$. Then $Q = F - T = \{Q_1, Q_2, \ldots, Q_{n-r}\}$ is used as a test set. Each entry $T_i$ in the training set is represented by the vector $T_i = (t_{i1}, t_{i2}, \ldots, t_{im})$, and each entry $Q_i$ in the test set is represented by a vector $Q_i = (q_{i1}, q_{i2}, \ldots, q_{im})$.

4.5.1 Data Processing

From Figure 4.5 and 4.6, we can see that the shapes of the function length pattern are similar within the same malware family and different across families. This is the motivation of using function length features in our experiments. While as I mentioned above the raw function length vectors are of different sizes so are not directly comparable. We need to standardize the original vectors so that the patterns could be compared. Two standardization approaches are adopted in our experiments,
which leads to two tests: the function length frequency test and the function length pattern test. The latter will be described in Section 4.6.

The function length frequency test is based on counting the number of functions in different length ranges. We divided the function length scale into intervals, which we call *bins*, and for each sample counted the frequency of functions occurring in each bin. Due to the order of magnitude of differences between function lengths, we increased the range covered by our bins exponentially. For example, we might count the number of functions of lengths between 1 and \( e \) bytes, the number between \( e \) and \( e^2 \) bytes, etc. In our experiment, we chose \( m = 50 \) as the number of bins. This now allows us to associate a new vector of size 50 with each function length vector in the population as described below. In introducing a factor to include the height variations, we map an exponential function over the entire spectrum of the dataset, from heights 1 to \( M \), the maximum function length across the complete dataset. Assuming that this exponential function is given by \( y = ae^{kx} \) where \( y(0) = 1 \) and \( y(50) = M \), it follows that \( a = 1 \), \( k = \ln M/50 \), and so \( y = e^{\ln M/50} \).

Thus, for any \( P_k \) from the population, the \( j \)th entry in the standardized form \( P_k^* \) of \( P_k \) of size 50 is:

\[
p^*_kj = \lvert \{p_{ki} \mid e^{\ln(M/50)(j-1)} \leq p_{ki} \leq e^{\ln(M/50)j}, \quad i = 1 \ldots m_k \} \rvert
\]

for \( j = 1 \ldots 50 \).

### 4.5.2 Statistical Test

We assume that the vectors of the entire population have been standardized as described in Section 4.5.1. For each family \( F \) we choose 80% as a training set from which we compute a single ‘centroid’ vector to use in comparing against the entire dataset.
as a means of classification. We obtain this centroid vector \( A = (a_1, a_2, \ldots, a_{50}) \) by computing each term as follows:

\[
a_j = \frac{1}{r} \sum_{i=1}^{r} t_{ij}, \ j = 1 \ldots 50.
\]  

(4.5.2)

\( r \) is the number of malware files in the training set, its value varies from family to family due to we choose 80% malware files from that family as a training set. For each family, this process was repeated five times, each time using a different 80% of the family and in such a way that each vector appears in exactly one test set.

For each training set, the complement within the family is used as the test set. The Chi-square test is applied as a test for Goodness of Fit of the centroid vector and vectors in the training set [Spa05]. For each \( T_i = (t_{i1}, t_{i2}, \ldots, t_{i50}) \) in the training set, a Chi-square vector \( \chi^2 = (\chi^2_1, \chi^2_2, \ldots, \chi^2_{50}) \) is computed as

\[
\chi^2_j = \frac{(t_{ij} - a_j)^2}{a_j}, \ j = 1 \ldots 50.
\]  

(4.5.3)

Finally, \( \chi^2 \) is compared against a threshold value \( \epsilon \) from a standard Chi-square distribution table [Spa05]. A significance level of 0.05 was selected, which means that 95% of the time we expect \( \chi^2 \) to be less than or equal to \( \epsilon \). For each \( T_i \), let

\[
U_i = \{ t_{ij} | \chi^2_j \leq \epsilon, \ j = 1 \ldots 50 \}.
\]  

(4.5.4)

For each \( T_i \), the value \( \lambda_i \) defined by:

\[
\lambda_i = \frac{|U_i|}{50}
\]  

(4.5.5)

represents the proportion of components of \( T_i \) which fall within the threshold \( \epsilon \). Thus

\[
\lambda_A = \frac{1}{r} \sum_{i=1}^{r} \lambda_i
\]  

(4.5.6)

represents the proportion of elements of the training set which fall within the threshold.
We now apply this test to the set of standardized vectors from the entire dataset, excluding those used in the training set. Let $\mathbf{T}^*$ be the set of adjusted vectors from $\mathbf{T}$ as in Equation (1). Let $X$ now be any vector from the set $\mathbf{P}^* - \mathbf{T}^*$. We compare $X$ with $A$ by applying Equations (4.5.3) and (4.5.4) to produce $\lambda_X$ as in Equation (4.5.5).

Let $P(F, A)$ represent the set of vectors which were classified by our test as belonging to the family. It is constructed as follows:

$$X \in P(F, A) \text{ iff } \lambda_X \geq \lambda_A.$$  \hfill (4.5.7)

We repeat this process for all five training sets of each family. Every time a centroid vector $A$ is obtained for a specific training set of the family.

Table 4.2 in the next subsection summarizes the classification accuracy of our tests.

### 4.5.3 Test Results

<table>
<thead>
<tr>
<th>FAMILY</th>
<th>P</th>
<th>N</th>
<th>Accuracy</th>
<th>TPRATE</th>
<th>FPRATE</th>
<th>FNRATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLANGER</td>
<td>47</td>
<td>3370</td>
<td>0.9699</td>
<td>0.8085</td>
<td>0.0279</td>
<td>0.1915</td>
</tr>
<tr>
<td>ROBNOT</td>
<td>101</td>
<td>3100</td>
<td>0.9884</td>
<td>0.6337</td>
<td>0</td>
<td>0.3663</td>
</tr>
<tr>
<td>ROBOZIPS</td>
<td>82</td>
<td>3195</td>
<td>0.9774</td>
<td>0.7683</td>
<td>0.0172</td>
<td>0.2317</td>
</tr>
<tr>
<td>ALUREON</td>
<td>43</td>
<td>3390</td>
<td>0.6685</td>
<td>0.7209</td>
<td>0.3322</td>
<td>0.2791</td>
</tr>
<tr>
<td>BAMBO</td>
<td>41</td>
<td>3400</td>
<td>0.6493</td>
<td>0.6098</td>
<td>0.3503</td>
<td>0.3902</td>
</tr>
<tr>
<td>BEOVENS</td>
<td>144</td>
<td>2885</td>
<td>0.9243</td>
<td>0.5417</td>
<td>0.0565</td>
<td>0.4583</td>
</tr>
<tr>
<td>BOXED</td>
<td>263</td>
<td>2290</td>
<td>0.9651</td>
<td>0.6635</td>
<td>0.0004</td>
<td>0.3347</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>0.8776</td>
<td>0.6783</td>
<td>0.1121</td>
<td>0.3217</td>
</tr>
<tr>
<td>W.Average</td>
<td></td>
<td></td>
<td>0.9263</td>
<td>0.6574</td>
<td>0.0549</td>
<td>0.3426</td>
</tr>
</tbody>
</table>

Table 4.2. Function Length Frequency Results

For each family $F$ in Table 4.1, $TP$ represents the true positives, that is the number of samples belonging to the family which our test correctly classified. Formally,
\[ TP = |Q \cap P(F,A)|, \] represents the number of samples in \( Q \) which were placed in \( P(F,A) \) for any of the five centroid vectors \( A \) from five tests. \( TN \) represents the true negatives, that is the number of samples not in \( F \) which were not placed in \( P(F,A) \) for all five centroid vectors \( A \). Similarly, \( FP \) represents the false positives, the number of samples not in \( F \) which were placed in \( P(F,A) \) by any centroid \( A \), while \( FN \) represents the false negatives, the number of samples in \( Q \) which were not placed in \( P(F,A) \) by some centroid \( A \). The total number of positives, \( P = TP + FN \), is the set of elements of \( Q \) repeated five times, one for each centroid, while the total negatives, \( N = TN + FP \), is the set of elements not in \( F \), again, repeated five times. Finally, the True Positive and False Positive rates are generated over the whole population and all five tests (per family) as in Equation (4.5.8).

\[
\begin{align*}
TPrate &= TP/P \\
FPrate &= FP/N \\
FNrate &= FN/P \\
Accuracy &= (TP + TN)/(P + N)
\end{align*}
\]

Equation (4.5.8) is the same as the Equation (3.7.1) mentioned in Section 3.7 of Chapter 3. The Accuracy in Equation (4.5.8) measures how closely the test determines true containment, or not, in the family. We thus expect it to be close to 1. While this is the case in Table 4.2, the average True Positive rate is a little disappointing. This motivated us to continue to the test described in Section 4.6.

### 4.6 Function Length Pattern Test

The graphs of function lengths of malware samples described in Section 4 appear to have some similarities within families and differences across families. In order to compare these graph patterns, we need to standardize the original function length
vectors of different sizes. In function length frequency test in Section 4.5, we standardized the original vectors by counting the number of functions in different length ranges. In this section, we again use function length as a distinguisher, but using a different approach to standardize the original vectors.

In this test we directly use the pattern made by the function length. We use two steps in order to prepare the data. First, we standardize the vector size across the entire dataset by resizing the function length vector along the x-axis by a rational factor. Each term in the new vector is a weighted average of corresponding terms in the old vector. In the second step, we retain the shape of the pattern of function lengths by standardizing along the y-axis family by family. We do this by multiplying each component of the old vector by a formula derived by averaging the first component (the shortest function length) and last component (the longest function length). The standardization of vector size must be made across the whole database as we need to compare all vectors pairwise; the standardization of height is done family by family as this appears to be a significant identifier.

4.6.1 Data Processing

Step 1: standardize vectors in the complete dataset

We first obtain the average size over all vectors in the dataset \( \bar{m} = \frac{m_1 + m_2 + \ldots + m_N}{N} \).

Then for each arbitrary \( P_K \) we standardize it by using the continuous function \( f \) defined over the domain \([0, m_k]\) given by:

\[
 f(x) = \begin{cases} 
  p_{k1}, & 0 \leq x < 1 \\
  p_{k2}, & 1 \leq x < 2 \\
  \vdots \\
  p_{km_k}, & m_k - 1 \leq x < m_k 
\end{cases} \tag{4.6.1}
\]
We create the new vector $\bar{P}_k$ of length $\bar{m}$, by dividing the domain of $f$ into $\bar{m}$ equal sections and calculating the mean value of $f(x)$ over each section of the domain. That is,

$$\bar{p}_{kj} = \frac{1}{C} \int_{(j-1)C}^{jC} f(x)dx$$

(4.6.2)

where $C = \frac{m_k}{\bar{m}}$ and $j \in \{1, 2, ..., \bar{m}\}$.

Each point in the function length vector $P_k$ is given equal representation in the resized vector $\bar{P}_k = (\bar{p}_{k1}, \bar{p}_{k2}, ..., \bar{p}_{km_k})$.

**Step 2: standardize pattern height family by family**

We choose a training set $T$ from a family $F$ (refer to Section 4.5 for the notation). Let $\bar{P}_k \in T \subset F$. We average over the first component and then over the last component in each vector $P_k \in T$. Let $v_1 = \frac{1}{n}(\bar{p}_{11} + \bar{p}_{21} + ... + \bar{p}_{n1})$, and $v_\bar{m} = \frac{1}{n}(\bar{p}_{1\bar{m}} + \bar{p}_{2\bar{m}} + ... + \bar{p}_{n\bar{m}})$. We obtain the $j$th entry in the standardized vector $P^*_k = (p^*_k1, p^*_k2, ..., p^*_km_k)$ from $\bar{P}_k$ by:

$$p^*_{kj} = \frac{(v_\bar{m} - v_1)(\bar{p}_{kj} - \bar{p}_{k1}) + v_1}{(\bar{p}_{km_k} - \bar{p}_{k1})}, \quad j = 1 \ldots \bar{m}.$$ 

(4.6.3)

At this point, all vectors in the population have been standardized for size and each family has been standardized for pattern.

**4.6.2 Statistical Test**

Using the similar method described in Section 4.5.2, for each family we choose an 80% subset as a training set from which we compute a single centroid vector to use in comparing against the entire dataset as a means of classification. We assume that
all vectors have been standardized as in Section 4.6.1. We again run this test five
times, each time using a different 80% portion of the family. For each test and each
family, we calculate \( v_1 \) and \( v_m \) using the method above. Using Equation (4.5.2) of
Section 4.5.2, we compute the centroid vector \( A = (a_1, a_2, \ldots, a_m) \).

In [Wei02] the author states that for large samples, the assumption of the parent
population being normally distributed is not needed for the Student t-test. Assuming
the whole dataset has a t-distribution, we therefore use the Student t-test to
compare the centroid vector with a sample vector. The standard deviation vector
\( S = (s_1, s_1, \ldots, s_m) \) is calculated by:

\[
s_j = \sqrt{\frac{\sum_{i=1}^{k} (t_{ij} - a_j)^2}{k}} \tag{4.6.4}
\]

For each \( T_i \) in the training set and for each component \( t_{ij} \) we calculate the \( t \)-value
to test whether the component’s value is consistent with belonging to the family. For
each \( T_i \), we get \( \tau_i = (\tau_{i1}, \tau_{i2}, \ldots, \tau_{im}) \) using the following formula:

\[
\tau_{ij} = \left| \frac{t_{ij} - a_j}{s_j} \right|. \tag{4.6.5}
\]

We chose a confidence level \( \alpha = 0.05 \) which means we expect that 95% of the
values are within \( \epsilon \) standard deviation of the centroid vector. Thus if \( \tau_{ij} \leq \epsilon \), this
component is consistent with belonging to the family. In our experiment, the number
of samples in each family is different, so we adjust \( \epsilon \) according to the size of each
family. For each \( T_i \), let

\[
U_i = \{ t_{ij} | \tau_{ij} \leq \epsilon, j = 1 \ldots m \}. \tag{4.6.6}
\]

Then we get the degree of membership \( \lambda_i \) from Equation (4.5.5) and the threshold
\( \lambda \) for the family from Equation (4.5.6). We thus acquire both the centroid vector \( A \)
and a threshold \( \lambda \) for each family. Based on these, we calculate the true positive and
false positive rates as in Equation (4.5.8).
4.6.3 Test Results

Table 4.3 presents the statistical analysis results of the function length pattern test.

<table>
<thead>
<tr>
<th>FAMILY</th>
<th>P</th>
<th>N</th>
<th>Accuracy</th>
<th>TPRATE</th>
<th>FPRATE</th>
<th>FNRATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAGGER</td>
<td>47</td>
<td>3370</td>
<td>0.6128</td>
<td>0.9361</td>
<td>0.3917</td>
<td>0.0639</td>
</tr>
<tr>
<td>ROBKNOT</td>
<td>101</td>
<td>3100</td>
<td>0.8688</td>
<td>0.8416</td>
<td>0.1303</td>
<td>0.1584</td>
</tr>
<tr>
<td>ROBZIPS</td>
<td>82</td>
<td>3195</td>
<td>0.8276</td>
<td>0.8537</td>
<td>0.1731</td>
<td>0.1463</td>
</tr>
<tr>
<td>ALUREON</td>
<td>43</td>
<td>3390</td>
<td>0.9849</td>
<td>0.6977</td>
<td>0.0115</td>
<td>0.3023</td>
</tr>
<tr>
<td>BAMBO</td>
<td>41</td>
<td>3400</td>
<td>0.8718</td>
<td>0.7805</td>
<td>0.1271</td>
<td>0.2195</td>
</tr>
<tr>
<td>BEOVENS</td>
<td>144</td>
<td>2885</td>
<td>0.7821</td>
<td>0.8542</td>
<td>0.2215</td>
<td>0.1458</td>
</tr>
<tr>
<td>BOXED</td>
<td>263</td>
<td>2290</td>
<td>0.6722</td>
<td>0.9126</td>
<td>0.3555</td>
<td>0.0874</td>
</tr>
</tbody>
</table>

Table 4.3. Function Length Pattern Results

The True Positive rate achieved in this test is much higher than that in the previous test while the level of Accuracy was retained.

4.7 Running Times

<table>
<thead>
<tr>
<th>Family</th>
<th>No.Sample</th>
<th>ExportingTime(secs)</th>
<th>FLV GenerationTime(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bambo</td>
<td>41</td>
<td>726</td>
<td>144</td>
</tr>
<tr>
<td>Boxed</td>
<td>263</td>
<td>14526</td>
<td>13618</td>
</tr>
<tr>
<td>Alureon</td>
<td>43</td>
<td>1956</td>
<td>603</td>
</tr>
<tr>
<td>Beovens</td>
<td>144</td>
<td>499</td>
<td>139</td>
</tr>
<tr>
<td>Robknot</td>
<td>101</td>
<td>2602</td>
<td>110</td>
</tr>
<tr>
<td>Clagger</td>
<td>47</td>
<td>490</td>
<td>24</td>
</tr>
<tr>
<td>Robzips</td>
<td>82</td>
<td>2683</td>
<td>501</td>
</tr>
</tbody>
</table>

Table 4.4. Running Times in the Function Length Experiments

In table 4.4, running times, including ExportingTime and FLV GenerationTime, are listed for each family. All the times in this table are in seconds. In the preprocess
phrase, all the samples are exported to ida2DBMS schema. The time spent on exporting executables to the ida2DBMS is the ExportingTime. FLV GenerationTime is used to generate function length vectors by database programs described in Section 4.3.3. Other running times are the execution times of algorithms in the experiments. Compared with ExportingTime and FLV GenerationTime, execution times are relatively negligible due to the simplicity of the algorithms. So the total running time for the 721 samples is the sum of ExportingTime and FLV GenerationTime, which is 38621 seconds.

4.8 Discussion

In the FLF test, the average true average positive rate is 67.83%. For “Clagger” family, we get the maximum value of true positive rate, which is 80.85%. And the average false positive rate is 11.21% and we get zero value of false positive rate for “Robknot” family which means that we can correctly classify samples which are not from “Robknot” family.

In the FLP test, the average true average positive rate is 83.95%. For “Clagger” family, we got the maximum value of true positive rate, which is 93.61%. And the average false positive rate is 20.15% and we got the minimum value of false positive rate for “Alureon” family, which is 1.15%.

The results of both FLF and FLP tests show the true positive rate to be much higher than the false positive rate. If function length contained no information we would expect the true positive rate and the false positive rate to be relatively equal. We can therefore conclude that function length contains statistically significant information in distinguishing between families of malware, and is therefore a useful component for consideration in a classification of malware [TBV08].
The function length pattern method correctly identified a higher proportion of the true positives compared with the frequency method. However, the frequency method had a lower false positive rate, giving it a higher overall accuracy. We believe that this is because in our FLP method we vary how close a test vector needs to be to the centroid vector depending on the Student t-distribution of the training set. This will broaden the accepted range for a family with a degree of variability. However the broader acceptable range is likely to also accept more false positives. From the point of view of classification accuracy, FLF test outperforms FLP test. So in our later experiments, we choose FLF as one of our static methods.

Our technique relies on unpacking. As I mentioned in Section 3.3.2.1 of Chapter 3, there are many methods and tools that we can use to unpack a packed executable. We choose VMUnpacker 1.3(For more information , please see Section 3.3.2.1 of Chapter 3) as our unpacking tool.

Our approach is efficient to execute and scalable. Our training and classification processes both execute in $O(n)$ time, where $n$ is the number of malware files. The feature extraction is leveraged by DBMS technology. Once we finish exporting the disassembling information from IDA Pro into the SQL Schema, we do not need access the executable files. And we can easily fetch function length information from our ida2DBMS schema by executing the four database functions or stored procedures described in Section 4.3.3.

The average classification accuracy of FLF test is 87.76% and the average classification accuracy of FLP test is 80.29%. In FLF test, we get the maximum value of classification accuracy for “Robknot” family, which is 98.84%, and in FLP test, we get the maximum value 98.49% of classification accuracy for “Alureon” family. These results verify Hypothesis 1 proposed in Section 2.4 of Chapter 2: It is possible to find static features which are effective in malware detection and classification. Function
length based features are simple and effective in malware detection and classification. While the target of our classification system is 97% classification accuracy, therefore function length alone is not sufficient. While taking the advantage of our DBMS technology, we can exploit more information stored in our ida2DBMS schema. In next Chapter 5 I present another feature PSI(Printable String Information) by using the same techniques.

4.9 Summary

In this Chapter, I presented our Function Length based static method. First I described IDA function and function length information, then explained the function length information extraction process. And finally I described two experiments: FLF test and FLP test in detail. And based on the experimental results I discussed the classification performance of these two methods.
Chapter 5

String Features based Methodology

In this Chapter, I present another static feature extraction and presentation method, which focuses on Printable String Information.

5.1 Introduction

As I mentioned in Section 2.3 of Chapter 2, malware analysis and extraction should be based on simple and effective feature sets and these feature sets would be easily extracted. In Chapter 4, I presented our function length based extraction and presentation method. PSI (Printable String Information) contains interpretable strings which carrying semantic interpretation of malicious behaviors and is easily extractable from unpacked malware, so we choose PSI as our next static feature. In this Chapter, I provide detailed description of PSI based static feature extraction and presentation.

The rest of the chapter is organized as follows: the relevant literature reviews are summarized in Section 5.2. In Section 5.3, I detail our analytical approach and data preparation and in Section 5.4 I describe the experimental set-up. Sections 5.5
present the test. In Section 5.7 I analyse and compare the results of PSI test and extended one and summarize in Section 5.8.

5.2 Related Work

In the research area of static analysis methodologies, many researchers used different information of binary code to extract different static features from unpacked executables without executing them to do the classification [Ghe05, KS06, SBN+10, PBKM07, XSCM04, XSML07, TBIV09, TBV08, WPZL09, YLCJ10, DB10, HYJ09]. These static information included the basic blocks of code in the malware, a function tree which is constructed from the control flow graph and sequences of API function calls.

In Chapter 4 I presented the function length features based static methodologies, in the following sections, I present our approach to the classification problem based on the static features extracted from printable string information.

5.3 Data Preparation

5.3.1 PSI

Printable String Information is a list of strings extracted from a binary along with the address at which each string resides [Eag08]. IDA recognizes a large number of string formats. If the bytes beginning at the currently selected address form a string of the defined style, IDA groups those bytes together into a string variable. Strings window is the built-in IDA equivalent of the strings utility which is used to display a list of strings extracted from a binary and the Setup Strings window is used to
configure the defined style of string. Table Strings_window in our ida2DBMS schema table is used to store these information.

5.3.2 Extract Printable String Information

![Diagram of PSI exporting process]

**Figure 5.1. Exporting of PSI**

Figure 5.1 illustrates the exporting of PSI in our system. Once IDA finishes disassembling of an executable file, it generate a .idb file for the executable. This .idb file reflects all the disassembling information and IDA no longer requires access to that executable. Every time the Strings window is opened, IDA String Scanner scans the .idb file in accordance with the settings configured by Setup Strings window. By default IDA searches for and formats C-style null-terminated strings, you can reconfigure the Setup Strings window to search for anything other than C-style strings. In our system, we use the default settings. Strings window is used to display these formatted strings. Figure 5.2 is a snapshot of Strings window in IDA. Then using our customized ida2sql mentioned in Section 3.3.2.2 of Chapter 3, these string information is exported to table Strings_window in our ida2DBMS schema.
5.4 Experimental Set-up

5.4.1 Motivation

From the experiences of anti-virus analysts from CA company, we know that there are similar printable strings in specific malicious executables that distinguish them from others. Furthermore, these printable strings contain interpretable strings, including API calls from Import Table and strings carrying semantic interpretation, are the high-level specifications of malicious behaviors and contain the important semantic information which can reflect the attacker’s intent and goal [YCW+09]. In addition, as I show in Chapter 4, function length contains statistically significant information in distinguishing between families of malware, however, function length alone is not sufficient. And our ida2DBMS contain all the disassembling information.
extracted from executables. There three factors motive us to extract PSI to do malware detection and classification.

5.4.2 Test Dataset

Table 5.1 lists our test dataset. In this experiment, we expand the test dataset to 1367 samples by introducing viruses, cleanware and more recent malware samples with an extended detection date to November of 2008. In this table the first 7 families have been pre-classified as Trojans; the next 3 families as Viruses.

Due to the huge amount of malware released in the wild every day, fast response is extremely important for the anti-virus detection engine. Therefore, knowing whether a file is malicious or clean is more urgent than knowing the specific family from which the malware is from. In order to verify the that our methods keep effective in differentiating malware from cleanware, we introduce cleanware into our experiments. In our experiments cleanware is a group of clean executables collected from win32 platforms spanning from Windows 98 to Windows XP. In this experiment, we treat the cleanware as a single family.

5.5 Printable String Information Test

Figure 5.3 outlines our PSI experiment. The methodology used in this experiment can be summarised in the following steps:

- **Step 1**: Extract the features based on string information.
- **Step 2**: Generate string list based on the whole dataset.
- **Step 3**: For each sample, check the occurrence of each string from the string list, mapping each sample into a vector.
<table>
<thead>
<tr>
<th>Type</th>
<th>Family</th>
<th>Detection Date: starting ⇒ ending (YYYY-MM)</th>
<th>No. of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trojan</td>
<td>Bambo</td>
<td>2003-07⇒2006-01</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Boxed</td>
<td>2004-06⇒2007-09</td>
<td>263</td>
</tr>
<tr>
<td></td>
<td>Alureon</td>
<td>2005-05⇒2007-11</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Beovens</td>
<td>2005-03⇒2007-06</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>Robknot</td>
<td>2005-10⇒2007-08</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Clagger</td>
<td>2005-11⇒2007-01</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Robzips</td>
<td>2006-03⇒2007-08</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal of Trojan</strong></td>
<td><strong>721</strong></td>
<td></td>
</tr>
<tr>
<td>Virus</td>
<td>Agobot</td>
<td>2003-01⇒2006-04</td>
<td>340</td>
</tr>
<tr>
<td></td>
<td>Looked</td>
<td>2003-07⇒2006-09</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Emerleox</td>
<td>2006-11⇒2008-11</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal of Virus</strong></td>
<td><strong>485</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Total of Malware</strong></td>
<td><strong>2003-07⇒2008-11</strong></td>
<td><strong>1206</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Cleanware</strong></td>
<td></td>
<td><strong>161</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1367</strong></td>
</tr>
</tbody>
</table>

Table 5.1. Experimental Set of 1367 Files

- **Step 4**: Build family $F_i, i = 1 \ldots 11$, one of these families is cleanware.

- **Step 5**: Select a family $F_i$ and split it into 5 parts.

- **Step 6**: Build 5 sets $T$ and $Q$ using an 80% to 20% split.

- **Step 7**: Reduce the features by retaining only the strings which occur in more than 10% of the samples in the training set $T$.

- **Step 8**: For each set $T$ and $Q$, take a random sample of executable files from the other families to create supplementary vectors which represent negative samples (i.e. not from family $F_i$). Add the negative sample vectors to sets $T$ and $Q$.

- **Step 9**: Select the classifier.

- **Step 10**: Call WEKA libraries to train the classifier using the training data $T$. 

120
• **Step 11:** Evaluate the test data using set $Q$.

• **Step 12:** Repeat steps 9 to 11 for each classifier.

• **Step 13:** Repeat steps 5 to 12 for each family.

And this experiment can be divided into the following three phases including:

1) **Data Preprocessing and Feature extraction.**

2) **Feature selection and data preparation.**

3) **Classification and Performance evaluation.**

In the following sections, I provide the detailed description of each phrase.

### 5.5.1 Data Preprocessing and Feature Extraction

The first phase “Data Preprocessing and Feature extraction” uses the same techniques described in Sections 3.3.2 and 3.4.1 of Chapter 3. We first unpack the malware using VMUnpacker, disassemble both unpacked malware and cleanware and export disassembling information into our ida2DBMS schema.

In this experiment, printable string information extracted from executable files is used as static features. For each file, we create a vector to represent which strings are present in that file. We first generate a global list of all the strings that occur in our malware database. (We ignored strings of less than three bytes as these tend to be extremely common non-malicious strings and they unnecessarily add complexity to our computations.). In our ida2DBMS schema, I create a database view to generate this global string list. Figure 5.4 shows how to fetch the global string list by executing a sql script. In the output window of the sql script, we can see that there are two
Part-1: Data Preprocessing and Feature extraction

- Extract String Information for each file.
- Create Family Data [F1,...,F11]
- Exporting to ida2DBMS

Part-2: Feature selection and data preparation

- Select Fi and split it 5 different folds
- Select same number of instances of Fi from F_{i+1..11} and split it into 5 different folds
- Create Training set (Tr_i) from Fi \cup F_{other} [-F_{i+1..11}]
- Create test set (Ts_i) from rest of the part both from Fi and F_{other}
- Data reduction from Tr_i Based on string frequency
- Repeat for j=1..5

It has been shown from our investigation that the % of feature reduction vary the classification performance

Part-3: Classification and Performance evaluation

- Create the data in WEKA format
- Select the classifier and train classifier using Tr_i
- Validate the Ts_i data
- Evaluate the performance of Fi
- Repeat for Fi+1..11

Repeat until i=0

Figure 5.3. Overview of PSI Experiment
columns in the list, one is the string, another column “Frequency” indicates the global frequency of the string in our database. Each string is given an ordering according to its frequency in the list. For each malware, we then test which of the strings in our global list it contains. This is computed efficiently by using a hash table to store the global list and by doing hash lookups on the strings contained in a given sample. The results are recorded in a binary vector, where for each string in the global list, a 1 denotes that it is present and a 0 that it is not.

Figure 5.4. Global String List

Let $G = \{s_1, s_2, s_3, \ldots s_{|G|}\}$ be the global ordered set of all strings; thus $s_i$ might be the string “kernel32”. $|G|$ is the total number of distinct string, which is 31217 in the experiment.

Let $F$ represent the ordered set of malware families along with a single family of clean files. In our experiment, we take cleanware as a separate family. So there are
total $n = 11$ families in the experiment.

$$F = \{F_1, F_2, \ldots, F_1\}.$$ (5.5.1)

Each malware family $F_i$ is a set of executable files. We represent an executable file in the database by its module id $M_j$. Thus, for any particular family $F_i$, we can write

$$f_i = \{M^i_1, M^i_2, \ldots, M^i_k\}$$ (5.5.2)

where $k$ is the number of executables in a particular family. We can also view $M^i_j$ as a binary vector capturing which of the strings in the global set $G$ are present in the sample; that is:

$$M^i_j = (s^i_{j1}, s^i_{j2}, \ldots, s^i_{j|G|})$$ (5.5.3)

where

$$s^i_{jl} = \begin{cases} 0 & \text{if the } l\text{th string of } G \text{ is not in module vector } M^i_j, \\ 1 & \text{otherwise} \end{cases}$$

$|G|$ represents the total number of distinct strings in this experiment, which is 31217 in this experiment.

To be simplify, let us assume that the global set $G$ contained only 5 strings: $G = \{\text{"kernel32"}, \text{"advapi32"}, \text{"GetModuleHandleA"}, \text{"OpenProcess"}, \text{"LoadLibrary"}\}$ then $M^i_j = (1, 0, 1, 0, 1)$ means the strings “kernel32”, ”GetModuleHandleA” and ”LoadLibrary” are all present in the sample $M^i_j$, whereas “advapi32” and “OpenProcess” are not.

### 5.5.2 Feature Selection and Data Preparation

The second phase “Feature selection and data preparation” is a process that selects a subset of the original features removing irrelevant, redundant or noisy data. It also improves the performance of data classification as well as speeding up the processing algorithm. We considered only the data in the training sets when we selected the features.
For any particular family $F_i$ containing $k$ malware files, $T = \{T_1, T_2, \ldots, T_r\} \subseteq F_i$ represents a training set chosen from the family $F_i$. Then $Q = F_i - T = \{Q_1, Q_2, \ldots, Q_{k-r}\}$ is used as a test set. Each entry $T_j$ in the training set is represented by the vector $T_j = (t_{j1}, t_{j2}, \ldots, t_{j|G|})$, and each entry $Q_j$ in the test set is represented by a vector $Q_j = (q_{j1}, q_{j2}, \ldots, q_{j|G|})$.

To select features for the family $F_i$ based on string information, we then selected a restricted ordered subset $G^r_i$ based on the global set $G$. That is -

\[ G^r_i = \{g^i_{r1}, g^i_{r2}, \ldots, g^i_{r|G^i|}\} \] (5.5.4)

where $g^i_{rj}$ are the strings from the training set $T_r$ for the family $F_i$ ordered as in $G$ when this string occurred with a frequency of at least 10% in $T_r$.

The training sets can also be represented by binary vectors of string information in the same way as the $M^j_i$ shown in equation (3); for a specific family $F_i$,

\[ T^i_j = (t^i_{j1}, t^i_{j2}, \ldots, t^i_{j|G^i|}) \] (5.5.5)

where

\[ t^i_{jl} = \begin{cases} 0 & \text{if the } l^{th} \text{ string of } G^i \text{ is not in module vector } T^i_j \\ 1 & \text{otherwise} \end{cases} \]

A frequency threshold of 10% was chosen. It reduced the number of strings in set $G$ from over 100,000 to a number in the 100s (depending on the family). Preliminary work showed that higher thresholds gave slightly better results for some of the classification algorithms. Determining the optimal frequency threshold is a potential topic for future investigation.

After reducing the set of strings to $G^r_i$ we added some negative samples to our sets $T$ and $Q$, i.e. samples that do not belong to family $F_i$. Negative samples were needed to train the classifier and for testing against false positives. We randomly
selected samples from the other families and created standardised vectors for each sample using our reduced set $G^r_v$.

5.5.3 Classification and Performance Evaluation

The third phrase “Classification and Performance evaluation” is to do classification using machine learning machine based method and to evaluate the performance of the system.

We built a program interface to WEKA to do classification. The program reads from the idb2DBMS database to collect the data for preprocess, feature extraction and feature selection. Then generates the training set $T$ and test set $Q$ and converts both sets into WEKA data format. We pass training set $T$ to the WEKA library to train the classifiers and then test the effectiveness with test set $Q$. Our program is designed in such a way that the system can select the families and the corresponding classifiers according to our requirements rather than the default in WEKA. And in this experiment we apply 5-fold cross validation in all cases.

We then use the same Equations (3.7.1) mentioned in Section 3.7 of Chapter 3 to evaluate the performance of our system. Table 5.2 and Table 5.3 give the results. And Figure 5.5 compares the classification accuracy with and without boosting for these five classification algorithms.

<table>
<thead>
<tr>
<th>Classification Algorithm</th>
<th>Base Classifier</th>
<th>Meta Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
</tr>
<tr>
<td>NB</td>
<td>0.015</td>
<td>0.1846</td>
</tr>
<tr>
<td>SVM</td>
<td>0.023</td>
<td>0.065</td>
</tr>
<tr>
<td>IB1</td>
<td>0.0266</td>
<td>0.072</td>
</tr>
<tr>
<td>RF</td>
<td>0.02</td>
<td>0.072</td>
</tr>
<tr>
<td>DT</td>
<td>0.041</td>
<td>0.162</td>
</tr>
</tbody>
</table>

Table 5.2. Average Family Classification Results in PSI Experiment
<table>
<thead>
<tr>
<th>Classification Algorithm</th>
<th>Base Classifier</th>
<th>Meta Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
</tr>
<tr>
<td>NB</td>
<td>0.0084</td>
<td>0.173</td>
</tr>
<tr>
<td>SVM</td>
<td>0.016</td>
<td>0.056</td>
</tr>
<tr>
<td>IB1</td>
<td>0.0175</td>
<td>0.0529</td>
</tr>
<tr>
<td>RF</td>
<td>0.011</td>
<td>0.051</td>
</tr>
<tr>
<td>DT</td>
<td>0.0291</td>
<td>0.167</td>
</tr>
</tbody>
</table>

Table 5.3. Weighted Average Family Classification Results in PSI Experiment

![Accuracy](image)

Figure 5.5. Comparison of Classification Accuracy (with and without Boosting)

### 5.5.4 Experimental Results

Table 5.2 presents the average of the experimental results according to classifier. Naïve Bayes gives the weakest results, while the other algorithms compare very well with each other. The meta-classifier AdaBoostM1 improves on all classifiers, with the exception of SVM, but the difference is insignificant. Based on these results, the best accuracy rate is above 96% (AdaBoostM1 with DT).

Since not all families were of the same size, we also calculated a weighted average, where each family was weighted according to the formula
where \( n_{fi} \) is the number of modules in family \( F_i \), and \( n_T \) is the total number of executable files (across all families). The weighted average results are shown in Table 5.3. For all parameters, the weighted results are better than the non-weighted results. The Random Forest and IB1 classifiers both achieve accuracies above 97%. Random Forest has the best results overall.

Again, AdaBoostM1 improves on all classifiers with the exception of SVM, but the difference is insignificant. The best accuracy rate after calculating the weighted average is 97.5% (AdaBoostM1 with RF).

As I mentioned in Section 5.4.2, we introduce cleanware in this experiment. Table 5.4 lists the classification results of malware versus cleanware by taking cleanware as a separate family.

<table>
<thead>
<tr>
<th>Classification Algorithm</th>
<th>Base Classifier</th>
<th>Meta Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Classifier</td>
<td>FP</td>
<td>FN</td>
</tr>
<tr>
<td>NB</td>
<td>0.04</td>
<td>0.2</td>
</tr>
<tr>
<td>SVM</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>IB1</td>
<td>0.15</td>
<td>0.068</td>
</tr>
<tr>
<td>RF</td>
<td>0.075</td>
<td>0.062</td>
</tr>
<tr>
<td>DT</td>
<td>0.16</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 5.4. Malware Versus Cleanware Results in PSI Experiment

### 5.6 Running Times

In the preprocess phase of static analysis, all the samples are exported to ida2DBMS schema. The time spent on exporting executables to the ida2DBMS is the ExportingTime. In Table 5.5, ExportingTimes are listed for the families and these times are in seconds. In ida2DBMS schema, I create a database view to update the
<table>
<thead>
<tr>
<th>Family</th>
<th>No.Samples</th>
<th>ExportingTime (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bambo</td>
<td>41</td>
<td>726</td>
</tr>
<tr>
<td>Boxed</td>
<td>263</td>
<td>14526</td>
</tr>
<tr>
<td>Alureon</td>
<td>43</td>
<td>1956</td>
</tr>
<tr>
<td>Beovens</td>
<td>144</td>
<td>499</td>
</tr>
<tr>
<td>Robknot</td>
<td>101</td>
<td>2602</td>
</tr>
<tr>
<td>Clagger</td>
<td>47</td>
<td>490</td>
</tr>
<tr>
<td>Robzips</td>
<td>82</td>
<td>2683</td>
</tr>
<tr>
<td><strong>Subtotal of Trojan</strong></td>
<td><strong>721</strong></td>
<td><strong>23483</strong></td>
</tr>
<tr>
<td>Agobot</td>
<td>340</td>
<td>48300</td>
</tr>
<tr>
<td>Looked</td>
<td>67</td>
<td>4062</td>
</tr>
<tr>
<td>Emerleox</td>
<td>78</td>
<td>7313</td>
</tr>
<tr>
<td><strong>Subtotal of Virus</strong></td>
<td><strong>485</strong></td>
<td><strong>45469</strong></td>
</tr>
<tr>
<td>Cleanware</td>
<td>161</td>
<td>7811</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1367</strong></td>
<td><strong>76763</strong></td>
</tr>
</tbody>
</table>

Table 5.5. Running Times in the Printable String Information Experiments

Global String List mentioned in Section 5.5.1, so the ExportingTimes already include the preprocess times. And the running times for the classification component of the experiments are 10269 seconds (estimated value based on the classification component of the experiment for static PSI features described in Section 8.6 of Chapter 8, the only difference between that experiment and this experiment is the number of samples. In that experiment, we test on 2939 samples and the running time of classification component is 368 mins)

### 5.7 Discussion

From these results in Table 5.2 and Table 5.3, we can see that the values of classification accuracy are all over 9% including Naïve Bayes which performs the worst. And for the weighted average classification results, all the values are over 95% with the exception of Naïve Bayes. Our PSI test results show that string information can be used to achieve high classification accuracy for the range of methods we tested. This is evidence that strings are a powerful feature for malware classification. Therefore
this experiment verifies again Hypothesis 1 proposed in Section 2.4 in Chapter 2: It is possible to find static features which are effective in malware detection and classification. In addition, perhaps surprisingly, many of the strings used for classifying came from library code (rather than the malicious code itself). This suggests that string information can be used to identify which libraries the programs used.

We think the reason of bad performance of Naïve Bayes in our experiments is the limitation of the classification algorithm. As I mentioned in Section 3.6.2.1 of Chapter 3, Naïve Bayesian classifiers are based on the assumption that all the attributes of the class are conditionally independent so the presence (or absence) of a particular attribute of a class is unrelated to the presence (or absence) of any other attribute. They consider all of the attributes to independently contribute to the probability summary of classification which may not be the case for our data. Results in Table 5.2 and Table 5.3 verify this hypothesis. In those experiments, Naïve Bayes gives the weakest results.

We introduced 161 clean executables collected from win32 platform in this experiment and Table 5.4 lists the classification results of malware versus cleanware by taking cleanware as a separate family. In that table, we can see that the best classification accuracy is 94.8% (RF with boosting). These results show that our methods keep similar performance in differentiating malware from cleanware.

In Table 5.6, we compare PSI results with some other recent work classifying large sets of malware and achieving at least 85% accuracy. Bailey et al. [BOA+07] describe malware behaviour in terms of system state changes (taken from event logs). They compare over 3000 pre-classified samples with over 3000 unclassified samples using a clustering technique to measure closeness. They achieve over 91% accuracy.

Rieck et al. [RHW+08] used behavioural pattern analysis to classify differences between 14 families of Trojans (3) and worms (11) based on a total of over 10,000
<table>
<thead>
<tr>
<th>Families</th>
<th>Size</th>
<th>Accuracy</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>K. Rieck et. al (2008) [RHW+08]</td>
<td>14</td>
<td>10,000</td>
<td>88% Behavioural patterns</td>
</tr>
<tr>
<td>M. Bailey et. al (2007) [BOA+07]</td>
<td>unknown</td>
<td>8,000</td>
<td>91.6% Behaviour</td>
</tr>
<tr>
<td>Z. Shafiq et. al (2009) [STF09]</td>
<td>unknown</td>
<td>1,200</td>
<td>95% This paper compares only clean with malicious files</td>
</tr>
<tr>
<td>Tian, R et. al. (2008) [TBV08]</td>
<td>7</td>
<td>721</td>
<td>87% Function length frequency</td>
</tr>
<tr>
<td>Our method</td>
<td>11</td>
<td>1,367</td>
<td>97% Printable string information</td>
</tr>
</tbody>
</table>

Table 5.6. Comparison of Our Method with Existing Work

samples. They used training, test and validation sets and applied SVM classifiers choosing the best such classifier family by family. Overall performance is determined by using a combined classifier on the testing partition. Approximately 88% true positive allocation is achieved. We also compare PSI test with our FLF test [TBV08].

5.8 Summary

In this Chapter, I presented our PSI (Printable String Information) based static method. First I explained the PSI and describe the extraction of PSI. Then I described the PSI test in its three phrases. And finally based on the experimental results I discussed the performance of our system.
Chapter 6

Combined Static Features based Methodology

6.1 Introduction

This chapter presents an automated malware classification system based on static feature selection. In previous Chapters 4 and 5, I presented two static classification methodologies FLF method and PSI method, where feature vectors are extracted individually based on function length frequency and printable string information. From the FLF experimental results of Chapter 4, we saw that some aspects of the global program structure remain consistent across malware families despite the evolution of the code. Thus function features of the unpacked malware are useful in identification. And from PSI experimental results of Chapter 5, we found that string information can be used to achieve high classification accuracy for the range of methods we tested.

As **Hypothesis 2** that I presented in Chapter 2, our expectation is that function length and printable strings are complementary features for classifying malware, we
expect each to reinforce the other, thus together giving better results than either separately. The purpose in this chapter is to test this hypothesis.

Thus, in this chapter I present a combined approach drawing on both the FLF and PSI methodologies. The combined results show that the malware classification accuracy achieves a weighted average of over 98% (with DT), which indeed improves on both methodologies separately and confirming our hypothesis. These results also strengthen the argument against the existence of a unique feature usable for malware identification.

The rest of the chapter is organized as follows: the relevant literature reviews are summarized in Section 6.2. In Section 6.3, I detail our analytical approach and data preparation and in Section 6.4 I describe the experimental set-up. Section 6.5 presents the test. In Section 6.6 I analyse and compare the results of the PSI test and combined one and summarize in Section 6.7.

6.2 Related Work

At the side of static features, many works have been done [Ghe05, KS06, SBN+10, PBKM07, XSCM04, XSML07, WPZL09, YLCJ10, DB10, HYJ09]. Different researchers used different aspects of binary code to do the classification, including basic blocks of code in the malware, a function tree which is constructed based on the control flow graph and sequences of API function calls. To the time of writing this thesis there was a lack of research work combining more than two kinds of static features together to improve the classification performance.

Based on our previous work [TBIV09, TBV08], in the following sections, I present our approach to the classification problem based on the Combined Static Features extracted from executables.

133
6.3 Data Preparation

In this experiment, we extend FLF method presented in Chapter 4 to the test dataset of 1367 samples listed in Chapter 5 and merge the extracted static features using both the FLF and PSI methods, then pass the generated vectors into the classification engine. Figure 6.1 gives an example of the combined feature vector.

In Chapter 4 we presented two function length related tests: FLF (Function Length Frequency) test and FLP (Function Length Pattern) test. From results of two tests, we saw that FLF method outperforms FLP method according to the classification accuracy. FLP method identified a higher proportion of the true positives compared with FLF method. However, FLF method had a lower false positive rate, giving it a higher overall accuracy. As I discussed in Section 4.8 of Chapter 4, this is because in our FLP method we vary how close a test vector needs to be to the centroid vector depending on the Student t-distribution of the training set. This will broaden the accepted range for a family with a degree of variability. However the
broader acceptable range is likely to also accept more false positives. Furthermore
the target of our system is to achieve a higher classification accuracy, so we choose
FLF method and combine it with PSI method in this experiment.

6.4 Experimental Set-up

6.4.1 Motivation

As I proposed in Chapter 2, our Hypothesis 2 is that function length and
printable strings are complementary features for classifying malware, we expect each
to reinforce the other, thus giving better results than either separately.

In Chapter 2, I talk about signature-based malware detection system which relies
on the determined signature. The implication is that there is some unique factor
which defines a piece of code. While this may be the case for a specific sample,
given the many obfuscation techniques available, it is unlikely to be true for a general
family; there may be several features of a piece of code which together indicate its
purpose, but which separately do not definitively reveal this information.

Our results in Chapter 4 and 5 verify our Hypothesis 1 proposed in Chapter 2.
FLF (function length frequency) and PSI (printable string information), these simple
static features extracted from binary code are effective in malware detection and
classification. In this experiment, we expect to verify Hypothesis 2 by combining
these two static methods in our malware detection and classification system.

6.4.2 Test Dataset

In this experiment, we use the same test dataset as our PSI experiment, see
Table 5.1 in Chapter 5. There are 1367 executables , including trojans, viruses and
cleanware. The first 7 families in Table 5.1 have been pre-classified as trojans; the
next 3 families as Viruses.

6.5 Combined Test

6.5.1 Overview of Experimental Process

Figure 6.2 illustrates the process of our Combined Static Features Based Classifi-
cation experiment. In this experiment, we use the \textit{K-fold} cross validation technique
with $K = 5$. The methodology used can be described by the following steps:

- \textbf{Step 1} Extract the features based on both FLF and PSI methods.

- \textbf{Step 2} Combine the features vectors and make a WEKA (arff) file.

- \textbf{Step 3} Build families $F_i$ for $i = 1 \ldots 11$.

- \textbf{Step 4} Select a family $F_i$ and split it into 5 parts.

- \textbf{Step 5} Select the same instances of $F_j$ from other families $F_j (j \neq i)$ and split
into 5 parts in the same manner as for $F_i$.

- \textbf{Step 6} Build 5 sets of \textit{Training Set Data} and \textit{Test Set Data} in respective
portions of 80\% and 20\%.

- \textbf{Step 7} Select features from the \textit{Training Data}.

- \textbf{Step 8} Select the classifier $C$ to be used.

- \textbf{Step 9} Call WEKA libraries to train the $C$ using the \textit{Training Data}.

- \textbf{Step 10} Evaluate the \textit{Test Set Data}. 136
• **Step 11** Repeat four more times.

• **Step 12** Repeat for other classifiers.

• **Step 13** Repeat for other classifiers.

---

![Diagram](image)

**Figure 6.2. Combined Static Features Based Classification Process**

### 6.5.2 Experimental Results

In this experiment, we take the cleanware as a family and list the classification accuracy for cleanware family separately. It is shown that we achieve over 96% accuracy (meta classifier with DT) in our testing, as indicated in Table 6.1 and 6.2.

<table>
<thead>
<tr>
<th></th>
<th>NB</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>W. Avg</td>
<td>94.31</td>
<td>97.77</td>
<td>97.63</td>
<td>98.15</td>
<td>96.69</td>
</tr>
<tr>
<td>Cleanfiles</td>
<td>84.51</td>
<td>92.4</td>
<td>85.64</td>
<td>91.04</td>
<td>94.4</td>
</tr>
</tbody>
</table>

Table 6.1. Classification Results for Base Classifier
6.6 Discussion

Combined Static Features Experiment results show that combined features, using both FLF and PSI methodologies, can be used to achieve higher classification accuracy for the range of methods we tested compared to the previous results in Chapters 4 and 5, which were 87% and 97% respectively. This evidence supports our Hypothesis 2 that Combining several static features can produce better detection and classification performance than any individual feature can produce.

Figure 6.3 graphically represents the comparisons of classification accuracy with and without boosting. AdaBoostM1 improves on all classifiers although the difference is insignificant. Like the results in Section 5.5.4 of Chapter 5, Naïve Bayes gives the weakest results, while the other algorithms compare very well with each other. We think the reason of bad performance of Naïve Bayes in this combined experiment and previous PSI experiment is the limitation of the classification algorithm which I discussed in Section 3.6.2.1 of Chapter 3. Based on the weakest performance given by Naïve Bayes in both PSI and Combined methods, we then decide to exclude Naïve Bayes algorithm from our selected classification algorithms in the later experiments.

Based on these results, the best accuracy rate of the combined method is above 98.86% (AdaBoostM1 with DT).

Figure 6.4, I compare this combined classification method with the previous individual PSI classification method in Chapter 5 (which had better results than FLF.
method in Chapter 4). It is obvious from the figure that our present accuracy is better in all parameters compared to existing one.

![Accuracy Chart](chart.png)

**Figure 6.3. Comparison of Classification (with and without boosting)**

In Table 6.3, I compare our results with some other recent work classifying large sets of malware and achieving at least 80% accuracy. As different authors use different datasets, therefore a direct comparison is not possible, as there is no publicly available standardized malware test set.

Zhao, H., et al. [ZXZ+10] used string information to classify malware based on a total of over 13330 samples. They used training, test and validation sets and applied SVM classifiers choosing the best such classifier family by family. Overall performance is determined to be approximately 83.3%. Ye, Y. et.al [YLJW10], used a post-processing technique based on analysis of API execution calls. They adapt several post-processing techniques of associative classification in malware detection on approximately 35000 malicious and 15000 benign samples. Using various data mining techniques they achieved approximately 88% accuracy. I also compare our published results from Chapter 4 and 5, Ahmed et al. [AHSF09] whose aim is a composite
scheme for malware classification that extracts statistical features from both spatial and temporal information available in run-time API calls. Using 237 core API calls from six different functional categories, their system provides an accuracy of 98% on the average. They also carried out a scalability analysis with an accuracy of 97%. Their result is as good as our method in classification accuracy, while they test on a relatively small dataset with 416 samples, our dataset is more than three times of their size.

6.7 Summary

In this Chapter, the Combined Static Features experiment was presented. First I proposed our hypotheses that FLF and PSI can be combined to complement each other, making our system flexible, scalable and robust. Then I explained the Combined Static Features experiment and analyse the results. We did not set up an extended experiment for our expanded test dataset mentioned in the table 3.1 in Section 3.3.1 of Chapter 3. We established that the best classification accuracy rate
<table>
<thead>
<tr>
<th>Families</th>
<th>Size</th>
<th>Accuracy</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>[AHSF09]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ye, Y. et.al., (2010)</td>
<td>unknown</td>
<td>35000</td>
<td>88% Behavioral</td>
</tr>
<tr>
<td>[YLJW10]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hengli Zhao, et.al., (2010)</td>
<td>unknown</td>
<td>13332</td>
<td>83.30% String information</td>
</tr>
<tr>
<td>[ZZL09]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tian, R et.al. (2008)</td>
<td>7</td>
<td>721</td>
<td>87% Function length</td>
</tr>
<tr>
<td>[TBV08]</td>
<td></td>
<td></td>
<td>frequency</td>
</tr>
<tr>
<td>Tian, R et. Al (2009)</td>
<td>11</td>
<td>1367</td>
<td>97% Printable string</td>
</tr>
<tr>
<td>[TBIV09]</td>
<td></td>
<td></td>
<td>information</td>
</tr>
<tr>
<td>Our method</td>
<td>11</td>
<td>1367</td>
<td>98.86% Function length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp; printable string</td>
</tr>
</tbody>
</table>

Table 6.3. Comparison of Our method with Similar Existing Work

98.86% was high enough for static methods compared with our target of 97% classification accuracy and we will move to dynamic method and do the extended experiment by combining static and dynamic methods.
Chapter 7

Dynamic Methodology

This chapter proposes an approach for detecting and classifying malware by investigating the behavioral features using dynamic analysis of malware binaries. Our framework includes statistical methods for classifying malware into families using behavioral analysis of information collected at run-time.

7.1 Introduction

In Chapter 2, we proposed our integrated method which aims to build a robust malware detection and classification system by merging static and dynamic methods together to complement each other. In Chapters 4, 5 and 6 I elaborated the static methods. In this Chapter I present our approach for detecting and classifying malware by investigating the behavioral features using dynamic analysis of malware binaries. We have developed a fully automated tool called “HookMe” mentioned in Chapter 3 in a virtual environment to extract API call features effectively from executables. Our experimental results with a dataset of nearly 3000 malware executables stretching cross a 8-year span, including trojans, worms and viruses, have shown an accuracy
of over 90% for classifying the malware families and an accuracy of over 95% in differentiating malware from cleanware.

The rest of the chapter is organized as follows: the relevant literature reviews are summarized in Section 7.2. In Section 7.3, I detail our analytical approach and data preparation and in Section 7.4 I describe the experimental set-up. Sections 7.5 present both Family Classification Test and Malware versus Cleanware Test. In Section 7.6 I analyse and compare the results of the Dynamic method and some other similar methods. And summarize in Section 7.7.

7.2 Related work

In Section 2.2 of Chapter 2, I pointed out that all malware analysis and extraction approaches can basically be categorized into two types: (i) based on features drawn from an unpacked static version of the executable file without executing the analyzed executable files [Ghe05, KS06, SBN+10, PBKM07, XSCM04, XSML07, TBIV09, TBV08, WPZL09, YLCJ10, DB10, HYJ09] and (ii) based on dynamic features or behavioral features obtained during the execution of the executable files [CJK07, WSD08, AHSF09, KChK+09, ZXZ+10].

Due to the limitations of Static Method discussed in Section 2.2.2 of Chapter 2, more and more researchers turned to working on dynamic analysis techniques to improve the effectiveness and accuracy of malware detection and classification.

Viewing the malware as a black box, Christodorescu et al in [CJK07] focused on the interaction between malware and the operating system, therefore using system calls as the building blocks of their technique. The authors in [WSD08] used dynamic analysis technologies to classify malware by using a controller to manage execution, stopping the execution after 10 seconds. In [AHSF09] the authors opened a
new possibility in malware analysis and extraction by proposing a composite method which extracts statistical features from both spatial and temporal information available in run-time API calls. A novel malware detection approach was also proposed in [KChK+09]; the authors focused on Host-based malware detectors because these detectors had the advantage that they could observe the complete set of actions that a malware program performed and it was even possible to identify malicious code before it was executed. In [ZXZ+10], the authors also proposed an automated classification method based on the behavioral analysis. They tested on 3996 malware samples and achieved an average classification accuracy of 83.3%.

7.3 Data Preparation

In the dynamic analysis, each file is executed under a controlled environment which is based on Virtual Machine Technology. Our trace tool name “’HookMe’” is used to monitor and trace the real execution of each file.

As I mentioned in Section 3.3.3 of Chapter 3, we set up a virtual machine environment by installing and configuring VMware Server 2.0.2. Then we create a new virtual machine installing Window XP Professional as guest operation system and disable networking. Before the start of dynamic analysis, a snapshot of the virtual machine is taken and we need to revert to this snapshot for every execution. In such a way, we assure that the virtual machine is rehabilitated every time.

Figure 3.9 of Chapter 3 illustrates this process. To automatically run the executables, we code our vmrun-based Dynamic Analysis Script which carries out the executions in the VMware environment.
7.3.1 Dynamic Analysis Script

For each malware execution, the following steps are taken automatically in the Dynamic Analysis Script:

1) Revert the virtual machine to its original snapshot. It actually overwrites the current virtual machine state with the original virtual machine state, so that all the changes made by the malware during its execution are lost.

2) Start the virtual machine to set up the virtual runtime environment.

3) Copy the executable file from Host to VM.

4) Run our HookMe trace tool to monitor and trace the execution of malware. HookMe monitors the state changes in the system and generates a log file.

5) In our experiments, we run each executable file for 30 seconds.

6) Stop the virtual machine.

7) Restart the virtual machine and then copy the generated log file from virtual machine to Host.

After the execution, for each executable we get a log file which reflects the behaviors of the malware in terms of API function calls. In Section 3.4.2 of Chapter 3 I give an example of such log files.

7.4 Experimental Set-up

7.4.1 Motivation

Chapters 4, 5 and 6 focused on the static features methodologies. As I analyse in Section 2.2.2 of Chapter 2, Static Feature Extraction has its advantages and dis-
advantages. On the positive side, Static Feature Extraction is low level time and resource consuming[Li04] and has a easily accessible form[Oos08]; while on the negative side, Static Feature Extraction is susceptible to inaccuracies due to obfuscation and polymorphic techniques[MKK07].

The aim in this chapter is to build a dynamic analysis system which achieves accuracy similar to that of the static analysis approach. The purpose in this chapter is to test our Hypothesis 3 presented in Chapter 2: It is possible to find dynamic features which are effective in malware detection and classification.

7.4.2 Test Dataset

In this experiment, we introduce more recent malware families and more cleanware to expand our test dataset listed in Chapter 4, 5 and 6. In the dynamic experiment, we terminate the execution of each malware file after 30 seconds then collect the log files from the virtual machine environment. For some of malware files, there are no log files generated during the execution. In this case, we exclude those malware files from our test dataset.

See Table 3.1 in Chapter 3 for detailed information of all the families in this experiment. There are 2939 executables, including trojans, viruses, worms and cleanware. The first 12 families have been pre-classified as trojans; the next 2 families as worms; another 3 families as viruses. Now our test dataset includes malware files stretch cross a 8-year span which is from 2003 to 2010.

We know that knowing whether a file is malicious or clean is more urgent than knowing the specific family from which the malware is from. In order to verify the that our methods keep effective in differentiating malware from cleanware, we firstly introduce cleanware into our test dataset in Chapter 5. But in that experiment and later experiment presented in Chapter 6 we treat the cleanware as a separate family.
when we do classification. In order to differentiate malware from cleanware, we add malware versus cleanware tests in this chapter.

### 7.5 Dynamic Feature Based Tests

**VMware Server**

**HookMe**

Log Files

Virtual Machine

Collect all log files and regroup

Generate string list and string frequency of a file

Create a hash table to store all the unique strings along with frequency

Generate the feature vectors using frequency

Figure 7.1. Overview of Dynamic Feature Based Experiment.

**Figure 7.1** gives an overview of our Dynamic Feature Based experiment.

After we generate log files for all the files in our test dataset, we collect all the log files and regroup them according to their families. For each log file, we generate string list and string frequency, next we create a hash table to store all the unique strings along with frequency and generate the feature vectors using frequency. Then we pass these vectors to our WEKA-Integrated Classification Engine.
As I mentioned above, we actually do two experiments by using dynamic features. One is Malware VS Cleanware Classification and the other one is Malware Family Classification.

7.5.1 Malware VS Cleanware Classification

In this section I introduce our method for comparing the behaviour of malicious executables versus clean files using log files.

7.5.1.1 Feature Extraction

In the malware versus cleanware experiment we follow recommendations of [MSF+08] in using an equal portion of malware and of cleanware in the training as this gives the best classification results. Since we have almost five times as many malware files as cleanware files, we needed to run (at least) five tests in order to incorporate all the files at least once into our testing. We create subsets with equal numbers of malware and cleanware files.

We then generate string lists for each group. Our feature extraction method is to separate out all the API call strings and their parameters which occur in the log files. We treat the API calls and their parameters as separate strings. First we read the strings from our log files and record each string into our global list. We use a hash table to store all strings along with their global frequency and file frequency; we call it as $f_s$ and $f_m$ respectively. The $f_s$ is the total number of occurrences of the string in all files and the $f_m$ is the number of occurrences of the string within a file. We then compare the strings in each file against the global list. Figure 7.2 shows a sample of extracted features from our log file for a sample of malware along with their corresponding $f_m$ and $S$. 
Figure 7.2. Sample Feature Sets of A Malware File

For each file we calculate $f_m$ for all the strings in the global list. We also keep track of the global frequencies. Table 7.1 gives an example of the global frequencies and file frequencies for a small set of strings. In our classification experiments we find that only information about the presence of a string is important, however including the number of occurrences of the string, $f_m$, do not improve the classification results. Therefore our final feature vectors for each file only recorded which of the strings in the global list is present in each file. In our feature vectors we used “1” to denote that a string is present and “0” to denote that a string is not present.

The following example clarifies our feature extraction process. Let us assume that we consider three files, with the strings extracted from the execution logs of these files listed as follows:

- File 1 = \{ GetProcAddress, RegQueryValueExW, CreateFileW, GetProcAddress, ...... \}.
- File 2 = \{ GetProcAddress, OpenFile, FindFirstFileA, FindNextFileA, CopyMemory, ...... \}.
- File 3 = \{ GetProcAddress, CreateFileW, CopyMemory, RegQueryValueExW, ...... \}.

<table>
<thead>
<tr>
<th>Extracted $f_m$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P6</td>
<td>26</td>
</tr>
<tr>
<td>P7</td>
<td>41</td>
</tr>
<tr>
<td>P8</td>
<td>27</td>
</tr>
<tr>
<td>P16</td>
<td>25</td>
</tr>
<tr>
<td>P17</td>
<td>25</td>
</tr>
<tr>
<td>P18</td>
<td>27</td>
</tr>
<tr>
<td>P19</td>
<td>1</td>
</tr>
<tr>
<td>P150</td>
<td>41</td>
</tr>
<tr>
<td>P151</td>
<td>1</td>
</tr>
<tr>
<td>P460</td>
<td>27</td>
</tr>
<tr>
<td>P461</td>
<td>5</td>
</tr>
<tr>
<td>P833</td>
<td>27</td>
</tr>
<tr>
<td>P834</td>
<td>9</td>
</tr>
</tbody>
</table>
Table 7.1 lists the global frequencies and the file frequencies which would be generated for the strings in this example.

The feature vectors for the example files would be:

- File 1 = \{1,1,0,0,0,0,0\}.

- File 2 = \{1,0,0,1,1,1,1\}.

- File 3 = \{1,0,1,0,0,1\}.

### 7.5.1.2 Classification Process

In our classification, we integrated the WEKA library with our system. We use *K-fold* cross validation for classifying between malware and cleanware as this is a standard, well understood and well accepted method of classification in a number of domains including malware analysis. In our previous experiments, we chose *5-fold* cross validation. As I discussed in Section 3.6.1 of Chapter 3, we adjust to *10-fold* cross validation in this experiment in order to increase the chance of learning all the relevant information in the training set.

We test four base classifiers from WEKA: Support Vector Machine, Random Forest, Decision Table and the Instance-based classifier, IB1. In addition, the meta-
classifier with AdaboostM1) is tested added to each of the base classifiers. As I discussed in Chapter 3, 5 and 6 that we decided to exclude Naïve Bayesian classifiers from our experiment due to their assumption limitation and poor performance in experiments in Chapter 5 and 6.

Figure 7.1 shows our classification methodology. Each malware file is tested with cross validation once whereas each cleanware file has cross validation performed on it five times in correspondence with each malware group. The methodology used in this experiment is given below:

- **Step 1**: Extract string information from the collected log files.
- **Step 2**: Divide the malware files into 5 subsets with size equal to that of the cleanware set.
- **Step 3**: Build a hash table to record the global and file string frequencies ($f_s$ and $f_m$).
- **Step 4**: Create feature vectors and construct the WEKA (arff) file.
- **Step 5**: Select cleanware files and split them 10 groups.
- **Step 6**: Select a malware group and split it 10 groups.
- **Step 7**: Build Training Data and Test Data sets.
- **Step 8**: Select the classifier $C_i$, [$i = 1 \ldots n$], $n = 4$ represents four selected classifiers.
- **Step 9**: Call WEKA libraries to train the $C_i$ using Training Data.
- **Step 10**: Evaluate the Test Data set.
- **Step 11**: Repeat until finish all 10 groups.
• **Step 12:** Repeat for other classifiers.

• **Step 13:** Repeat for other malware subsets.

### 7.5.1.3 Experimental Results

Table 7.2 shows the average of the experimental results for each classifier in our malware versus cleanware test.

The performance of our experiment, malware versus cleanware, shows RF (Random Forest) gives the best average performance which is 94.80% compared to other classifiers while IB1 shows lower performance, which is 81.89%.

Table 7.3 also shows the average classification results using meta-classifier AdaBoostM1 on top of base classifiers. The results show that the meta-classifier improves the accuracy for all classifiers tested except SVM. Based on these results, the best accuracy rate is 95.15% (AdaBoostM1 with RF) and the lowest accuracy is 84.93% (AdaBoostM1 with IB1). Figure 7.3 graphically depicts the comparison of classification accuracy of base classifiers with metaclassifiers. Our results show that classification of cleanware versus malware using behavioural features can be used to achieve high classification accuracy.

### 7.5.2 Malware Family Classification

In our second experiment we achieve further refinement of our classification by comparing malware across and within families. By comparing individual malware files based on their behavior, we determine similarities which indicate a commonality. As we shall show, the results of this work, to a large extent, validate the pre-classification within the CA zoo.
<table>
<thead>
<tr>
<th>Clean Vs Group</th>
<th>SVM FP</th>
<th>SVM FN</th>
<th>SVM Acc</th>
<th>IB1 FP</th>
<th>IB1 FN</th>
<th>IB1 Acc</th>
<th>DT FP</th>
<th>DT FN</th>
<th>DT Acc</th>
<th>RF FP</th>
<th>RF FN</th>
<th>RF Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean Vs Group1</td>
<td>0.029</td>
<td>0.065</td>
<td>0.9519</td>
<td>0.276</td>
<td>0.07</td>
<td>0.8129</td>
<td>0.061</td>
<td>0.137</td>
<td>0.8982</td>
<td>0.037</td>
<td>0.074</td>
<td>0.9426</td>
</tr>
<tr>
<td>Clean Vs Group2</td>
<td>0.032</td>
<td>0.076</td>
<td>0.9444</td>
<td>0.226</td>
<td>0.107</td>
<td>0.8222</td>
<td>0.059</td>
<td>0.091</td>
<td>0.9222</td>
<td>0.047</td>
<td>0.065</td>
<td>0.9417</td>
</tr>
<tr>
<td>Clean Vs Group3</td>
<td>0.042</td>
<td>0.072</td>
<td>0.9407</td>
<td>0.239</td>
<td>0.037</td>
<td>0.85</td>
<td>0.069</td>
<td>0.12</td>
<td>0.9019</td>
<td>0.027</td>
<td>0.063</td>
<td>0.9537</td>
</tr>
<tr>
<td>Clean Vs Group4</td>
<td>0.032</td>
<td>0.074</td>
<td>0.9454</td>
<td>0.259</td>
<td>0.085</td>
<td>0.8148</td>
<td>0.056</td>
<td>0.157</td>
<td>0.8907</td>
<td>0.045</td>
<td>0.061</td>
<td>0.9444</td>
</tr>
<tr>
<td>Clean Vs Group5</td>
<td>0.03</td>
<td>0.065</td>
<td>0.9509</td>
<td>0.29</td>
<td>0.093</td>
<td>0.7944</td>
<td>0.071</td>
<td>0.102</td>
<td>0.9102</td>
<td>0.03</td>
<td>0.052</td>
<td>0.9574</td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td>0.033</td>
<td>0.0704</td>
<td>0.9467</td>
<td>0.258</td>
<td>0.0784</td>
<td><strong>0.8189</strong></td>
<td>0.0632</td>
<td>0.1214</td>
<td>0.9046</td>
<td>0.0372</td>
<td>0.063</td>
<td><strong>0.9480</strong></td>
</tr>
</tbody>
</table>

Table 7.2. Base Classifiers on Malware Versus Cleanware Using Dynamic Method
<table>
<thead>
<tr>
<th>Clean Vs Group</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>Clean Vs Group1</td>
<td>0.039</td>
<td>0.059</td>
<td>0.9509</td>
<td>0.237</td>
</tr>
<tr>
<td>Clean Vs Group2</td>
<td>0.048</td>
<td>0.078</td>
<td>0.9370</td>
<td>0.178</td>
</tr>
<tr>
<td>Clean Vs Group3</td>
<td>0.046</td>
<td>0.076</td>
<td>0.9389</td>
<td>0.213</td>
</tr>
<tr>
<td>Clean Vs Group4</td>
<td>0.044</td>
<td>0.072</td>
<td>0.9417</td>
<td>0.217</td>
</tr>
<tr>
<td>Clean Vs Group5</td>
<td>0.046</td>
<td>0.063</td>
<td>0.9454</td>
<td>0.217</td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td>0.0446</td>
<td>0.0696</td>
<td>0.9428</td>
<td>0.2124</td>
</tr>
</tbody>
</table>

Table 7.3. Meta Classifiers on Malware Versus Cleanware Using Dynamic Method
This work can also be used in quickly identifying an unknown file by testing it against the known families. If the classification test identifies it as being a member of a family already considered malicious, then the file can be classified as malicious without further analysis.

7.5.2.1 Feature Extraction and Classification Process

The Feature Extraction process is similar to that described in Section 7.5.1.1. The only difference is that first we generate the string list family by family and then create a hash table for every family and store all string information along with the module frequency, into our hash table. For example, Let $G_i = \{s_{i1}, s_{i2}, s_{i3}, ... s_{in}\}$ be the set of distinct strings of the family $F_i$, where $s_{i1}, s_{i2}, ... s_{in}$ represents the string, for instance: $s_{i1} = "kernel32"$, $s_{i2} = "advapi32"$, $s_{i3} = "GetModuleHandleA"$, $s_{in} = "openProcess"$.

Each malware family $F$ contains a number of files. In particular, we can write:
\[ F = \{M_1, M_2, \ldots, M_q\} \]  
(7.5.1)

where each \( M_i \) is a file, \( q \) is the number of files in the family. We can also represent each file \( M \) as an array of 3-tuples of the form:

\[ M^j = \langle s, f(F, s), f(M, s) \rangle \]  
(7.5.2)

Where \( s \) is a string, \( f(F, s) \) is the total number of occurrences of string \( s \) within the family and \( f(M, s) \) is the number of occurrence of string within the file \( M \).

For each malware file, we then test which strings in our family list that it contains. After inserting all the strings along with corresponding values into the hash table, we follow the same process as described in Section 7.5.1.2. The process of malware family classification is similar to that described in Section 7.5.1.2. The only difference is in how we construct the training and test data. In this testing, we select a particular family and split them into \( K = 10 \) different groups (as in Figure 7.2). For training the classifiers, we choose an equal number of positive (from the family) and negative (non-family) files. We construct the negative set by randomly selecting files from the other families. We invoke WEKA libraries to train the classifier and evaluate the test data. The same rotating process applies for all other families.

### 7.5.2.2 Experimental Results

The experimental setup and the methodology of our family classification system is similar to that presented in Section 7.5.1.2. We tested four base classifiers: SVM, Random Forest, Decision Table and IB1, in our experiments. We used the same data set as listed in Table 3.1 in Chapter 3. Table 7.4 shows the accuracy of our family classification for the tested classifiers. This table indicates that RF has best performance and IB1 the worst.
As the families are not all of the same file size, we therefore calculated a weighted average, where each family was weighted according to the formula:

$$\frac{n_{f_i}}{n_T},$$

where \(n_{f_i}\) is the number of files in family \(f_i\) and \(n_T\) is the total number of files (across all families).

According to the weighted average, the RF classifier achieved the best overall results in our family classification with 90.4% accuracy, as shown in Table 7.5.

Table 7.4 and Table 7.5 also shows the malware family classification results using the meta-classifier AdaBoostM1 on top of the base classifiers. The results show that the meta-classifier achieves quite similar results compared with base classifier. Based on these results, the best accuracy rate is 90.4% (AdaBoostM1 with RF).

Figure 7.4 compares the classification accuracy of base classifiers with meta-classifiers.
7.6 Discussion

In dynamic test, the behavioral information from both malware and cleanware binary files are extracted to classify software in two different ways:

1) Malware versus Cleanware Classification.

2) Family Classification.

The extracted API calls form the basis for modelling the behavioural patterns of software in order to distinguish between classes. The first and foremost contribution of this method is the outlining of a methodology to extract relevant behavioral features of API calls; such features include hooking of the system services and creation or modification of files. The second contribution is the provision of a statistical analysis of the API calls from log files generated by executing the files in a virtual environment. The third contribution is a method of distinguishing malware from cleanware using a 2-class classification model; this model achieves over 90% performance in terms of
<table>
<thead>
<tr>
<th>Method</th>
<th>Experimental Data</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Santos et.al (2010) [SBN+10]</td>
<td>$6(F^{1})/13189(m^{2})/13000(c^{3})$</td>
<td>No accuracy provided.</td>
</tr>
<tr>
<td>Ye et.al(2010) [YLJW10]</td>
<td>35000(m)/15000(c)</td>
<td>88%</td>
</tr>
<tr>
<td>Wang et.al(2009) [WPZL09]</td>
<td>353(m)/361 (c)</td>
<td>93.71%</td>
</tr>
<tr>
<td>Moskovitch et al.(2008) [MSF+08]</td>
<td>7688(m)/22735(c)</td>
<td>95%</td>
</tr>
<tr>
<td>Our method</td>
<td>2398(m)/541(c)</td>
<td>95.15%</td>
</tr>
</tbody>
</table>

Table 7.6. Comparison of Our method with Existing Work

accuracy. The fourth contribution is malware family classification, a similar process to that of testing cleanware versus malware, and this model achieves a classification accuracy of over 95%.

### 7.6.1 Cleanware versus Malware Classification

Distinguishing malware from cleanware is more critical for malware detection than malware family classification, and can offer a first line of defence against malware. Recent research trends are moving toward the classification of clean files from malicious file [MSF+08, SBN+10, WPZL09, YLJW10]. We have investigated some recent work classifying malware versus cleanware and compare it with ours. Table 7.6 summarizes this comparison across results achieving at least 80% accuracy, and compares their outcomes with those of our method.

Santos et.al in [SBN+10] used opcode sequence frequencies as features to calculate the cosine similarity between two PE executable files. There are 13189 malware files from 6 malware families and 13000 benign files in their dataset. They show that if

---

1the number of malware families;  
2the number of malware files;  
3the number of clean files;
they select an appropriate similarity ratio threshold, their method would be a very useful tool to identify malware variants from benign files; they do not mention the classification accuracy. Ye et al. in [YLJW10] used static API call sequences as a feature of associative rule learning. They test 35000 malware and 15000 cleanware files and their classification accuracy is 88%. Wang et al. in [WPZL09] used static API call sequences from 714 files which are pre-identified as either normal (361) or virus programs (353); they obtain 93.71% classification accuracy. Moskovitch et al. in [MSF+08] used the text categorization process and examined the relationship between the MFP (malicious file percentage) in the test set, which represents real-life scenario, and in the training-set, which being used for training the classifier. They found that the best mean performance is associated with a 50% MFP in the training. Their result is as good as our method in classification accuracy by using static method, while we use dynamic method. As I discussed in Section 2.2.2 of Chapter 2, Both static methods and dynamic methods have their merits and limitations. Dynamic analysis is a necessary complement to static techniques as it is significantly less vulnerable to code obfuscating transformations. Further discussion is provided in Section 8.4.1 of Chapter 8.

7.6.2 Family Classifications

In our previous published work [TIBV10], we achieved an accuracy of over 97% by using this dynamic method with a test dataset of over 1500 malware stretching across 5-year span which is from 2003 to 2008, including trojans and viruses. In this experiment, we introduce worms and more recent malware families which are collected in 2009 and 2010. Table 7.7 lists the detailed family classification results in this experiments.

From Table 7.7 we can see that most of the families perform quite well and Rob-
knot achieves 100% accuracy. Old families collected between 2003 and 2008 performance better than the new families collected between 2009 and 2010. New families “Banker”, “Bancos” and “Adclicker” families give poor performance according to accuracy, which achieve 68.75%, 88% and 76.67% accuracy respectively.

As I discussed in Chapter 2, the disadvantages of dynamic analysis include limited view of malware, trace dependence and execution time period and lack of interactive behavior information. The “Banker” and “Bancos” families are trojans which attempt to steal sensitive information that can be used to gain unauthorized access to bank accounts via Internet Banking. They only perform malicious actions when they receive specific commands or Internet users perform certain actions, such as logging into specific banking websites. Since we run malware automatically with no human interaction, such behaviors will not be recorded in the log files, which is the likely cause of low accuracy for “Banker” and “Bancos” families.

“AdClicker” is a group of trojans that designed to artificially inflate the numbers of visitors to a given website by creating fake page views, to share the primary functionality of artificially generating traffic to pay-per-click Web advertising campaigns in order to create or boost revenue. “AdClickers” typically copy themselves to a system folder in an attempt to remain inconspicuous, and create a load point so that they run every time Windows starts. In our dynamic experiment, we disable the Internet connection and run each malware executable by reverting the VM to the original snapshot without restart the Windows. In this case, the malicious behaviors will not be recorded in the log files, which is the likely cause of low accuracy for “Adclicker” family.

The disadvantages of dynamic analysis mentioned above can perhaps be supplemented by integrate with static analysis methods. In Chapter 8, I will give a detailed description of the integrated experiment.
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th></th>
<th>Acc</th>
<th>IB1</th>
<th></th>
<th>Acc</th>
<th>DT</th>
<th></th>
<th>Acc</th>
<th>RF</th>
<th></th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP</td>
<td>FN</td>
<td></td>
<td>FP</td>
<td>FN</td>
<td></td>
<td>FP</td>
<td>FN</td>
<td></td>
<td>FP</td>
<td>FN</td>
<td></td>
</tr>
<tr>
<td>Adclicker</td>
<td>65</td>
<td>0.217</td>
<td>0.25</td>
<td>76.667</td>
<td>0.283</td>
<td>0.217</td>
<td>75</td>
<td>0.117</td>
<td>0.3</td>
<td>79.167</td>
<td>0.217</td>
<td>0.25</td>
<td>76.667</td>
</tr>
<tr>
<td>Bancos</td>
<td>446</td>
<td>0.127</td>
<td>0.111</td>
<td>88.068</td>
<td>0.136</td>
<td>0.1</td>
<td>88.182</td>
<td>0.13</td>
<td>0.125</td>
<td>87.273</td>
<td>0.095</td>
<td>0.1</td>
<td>90.227</td>
</tr>
<tr>
<td>Banker</td>
<td>47</td>
<td>0.4</td>
<td>0.225</td>
<td>68.75</td>
<td>0.325</td>
<td>0.375</td>
<td>65</td>
<td>0.45</td>
<td>0.3</td>
<td>62.5</td>
<td>0.475</td>
<td>0.275</td>
<td>62.5</td>
</tr>
<tr>
<td>Frethog</td>
<td>174</td>
<td>0.065</td>
<td>0.076</td>
<td>92.941</td>
<td>0.112</td>
<td>0.071</td>
<td>90.882</td>
<td>0.047</td>
<td>0.112</td>
<td>92.059</td>
<td>0.029</td>
<td>0.1</td>
<td>93.529</td>
</tr>
<tr>
<td>Gamepass</td>
<td>179</td>
<td>0.118</td>
<td>0.141</td>
<td>87.059</td>
<td>0.171</td>
<td>0.147</td>
<td>84.118</td>
<td>0.153</td>
<td>0.206</td>
<td>82.059</td>
<td>0.176</td>
<td>0.112</td>
<td>85.588</td>
</tr>
<tr>
<td>SillyAutorun</td>
<td>87</td>
<td>0.238</td>
<td>0.162</td>
<td>80</td>
<td>0.275</td>
<td>0.162</td>
<td>78.125</td>
<td>0.238</td>
<td>0.225</td>
<td>76.875</td>
<td>0.188</td>
<td>0.162</td>
<td>82.5</td>
</tr>
<tr>
<td>SillyDl</td>
<td>439</td>
<td>0.216</td>
<td>0.314</td>
<td>73.488</td>
<td>0.263</td>
<td>0.235</td>
<td>75.116</td>
<td>0.288</td>
<td>0.316</td>
<td>69.767</td>
<td>0.207</td>
<td>0.221</td>
<td>78.605</td>
</tr>
<tr>
<td>Vundo</td>
<td>80</td>
<td>0.038</td>
<td>0.088</td>
<td>93.75</td>
<td>0.162</td>
<td>0.075</td>
<td>88.125</td>
<td>0.062</td>
<td>0.088</td>
<td>92.5</td>
<td>0.05</td>
<td>0.112</td>
<td>91.875</td>
</tr>
<tr>
<td>clagger</td>
<td>44</td>
<td>0</td>
<td>0.025</td>
<td>98.75</td>
<td>0</td>
<td>0.025</td>
<td>98.75</td>
<td>0.025</td>
<td>0.025</td>
<td>97.5</td>
<td>0</td>
<td>0.05</td>
<td>97.5</td>
</tr>
<tr>
<td>agobot</td>
<td>283</td>
<td>0.019</td>
<td>0.006</td>
<td>98.75</td>
<td>0.025</td>
<td>0.006</td>
<td>98.438</td>
<td>0.012</td>
<td>0.012</td>
<td>98.75</td>
<td>0.016</td>
<td>0.019</td>
<td>98.281</td>
</tr>
<tr>
<td>allureon</td>
<td>41</td>
<td>0.075</td>
<td>0.075</td>
<td>92.5</td>
<td>0.15</td>
<td>0.075</td>
<td>88.75</td>
<td>0.125</td>
<td>0.025</td>
<td>92.5</td>
<td>0.15</td>
<td>0.1</td>
<td>87.5</td>
</tr>
<tr>
<td>bambo</td>
<td>44</td>
<td>0.017</td>
<td>0.017</td>
<td>98.333</td>
<td>0.067</td>
<td>0.033</td>
<td>95</td>
<td>0.033</td>
<td>0.033</td>
<td>96.667</td>
<td>0.05</td>
<td>0.05</td>
<td>95</td>
</tr>
<tr>
<td>boxed</td>
<td>178</td>
<td>0.025</td>
<td>0.011</td>
<td>98.214</td>
<td>0.032</td>
<td>0.029</td>
<td>96.964</td>
<td>0.014</td>
<td>0.014</td>
<td>98.571</td>
<td>0.018</td>
<td>0.029</td>
<td>97.679</td>
</tr>
<tr>
<td>emerleox</td>
<td>75</td>
<td>0.043</td>
<td>0.071</td>
<td>94.286</td>
<td>0.071</td>
<td>0.043</td>
<td>94.286</td>
<td>0.043</td>
<td>0.043</td>
<td>95.714</td>
<td>0.043</td>
<td>0.1</td>
<td>92.857</td>
</tr>
<tr>
<td>looked</td>
<td>66</td>
<td>0.017</td>
<td>0.017</td>
<td>98.333</td>
<td>0.083</td>
<td>0</td>
<td>95.833</td>
<td>0.05</td>
<td>0.033</td>
<td>95.833</td>
<td>0.05</td>
<td>0.033</td>
<td>95.833</td>
</tr>
<tr>
<td>robknot</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0.029</td>
<td>0</td>
<td>98.571</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>robzips</td>
<td>72</td>
<td>0.014</td>
<td>0</td>
<td>99.286</td>
<td>0.043</td>
<td>0</td>
<td>97.857</td>
<td>0.029</td>
<td>0</td>
<td>98.571</td>
<td>0.014</td>
<td>0</td>
<td>99.286</td>
</tr>
<tr>
<td>Cleanfiles</td>
<td>541</td>
<td>0.052</td>
<td>0.037</td>
<td>95.556</td>
<td>0.076</td>
<td>0.039</td>
<td>94.259</td>
<td>0.041</td>
<td>0.056</td>
<td>95.185</td>
<td>0.065</td>
<td>0.035</td>
<td>95</td>
</tr>
<tr>
<td>Total</td>
<td>2939</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.0934</td>
<td>0.0903</td>
<td>90.8184</td>
<td>0.1263</td>
<td>0.0907</td>
<td>89.1492</td>
<td>0.1048</td>
<td>0.1063</td>
<td>89.479</td>
<td>0.1024</td>
<td>0.0971</td>
<td>90.0237</td>
</tr>
<tr>
<td>Weighted Avg</td>
<td></td>
<td>0.0979</td>
<td>0.1051</td>
<td>89.8495</td>
<td>0.1271</td>
<td>0.0936</td>
<td>88.9657</td>
<td>0.1092</td>
<td>0.1212</td>
<td>88.4797</td>
<td>0.0971</td>
<td>0.0949</td>
<td>90.3989</td>
</tr>
</tbody>
</table>

Table 7.7. Detailed Family-wise Malware Classification Results Using Dynamic Method (Meta Classifiers)
We also compare our empirical performance of malware family classification with similar existing methods and summarised it in Table 7.8. We have considered only the recent results in Table 7.8 which achieved classification accuracy of 80% or above.

Sathyanarayan et al. in [SKB08] used static extraction to extract API calls from known malware in order to construct a signature for an entire class. The API calls of an unclassified file of malware can then be compared with the 'signature' API calls for a family to determine if the file belongs in the family or not. They tested eight families with 126 files in total, but no specific classification accuracy was provided. Ahmed et al. in [AHSF09] proposed a composite method which extracts statistical features from both spatial and temporal information available in run-time API calls. There are 516 malware files in their dataset and they obtained 96.3% classification accuracy. Compared with this, our method of using dynamic API features got accuracy over 90% with a relatively large collection of malware. And our method use string frequency features which are simple and easy to be extracted from log files. Wagener et al. in [WSD08] tested a small set of malware, 104 files, by using dynamic extraction technologies; they did not provide classification accuracy. The authors in [ZXZ+10] traced the behaviour of malware in a virtual machine environment and used these traces to extract string information. They then applied SVM classifiers and overall performance was determined to be approximately 83.3%.

7.7 Summary

The Dynamic method was proposed in this chapter. First I described the data preparation by narrating the Dynamic Analysis Script which is used to automatically execute the executables and collect the log files. Following this I explained the moti-
<table>
<thead>
<tr>
<th>Paper</th>
<th>Exp. Data</th>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
</table>
| Sathyanarayan et.al (2008)  | 8(F) 1  
126(S) 5 | Static | Did not mention the quantitative accuracy |
| Tian, R et.al. (2008)       | 7(F) /  
721(S) | Static | 87%                |
| Tian, R et.al. (2009)       | 13(F) /  
1367(S) | Static | 97%                |
| Ahmed et.al. (2009)         | 516(S)    | Dynamic | 96.30%             |
| Wagener et.al. (2008)       | 104(S)    | Dynamic | Did not mention the quantitative accuracy |
| Zhao et.al. (2010)          | 11(F) /  
3996(S) | Dynamic | 83.30%             |
| Our method                  | 2939(S)   | Dynamic | 90.40%             |

Table 7.8. Comparison of Similar Existing techniques with Our method

Evaluation of the dynamic tests and provided the dataset tested in the experiments. Next I elaborated two dynamic tests: Malware versus Cleanware Classification and Family Classification. I also compared our method with other similar works.
Chapter 8

Integrated Static and Dynamic Features

8.1 Introduction

In the previous chapters, I explained our static features based methods, including FLF method, PSI method and combined static features approach, and I also explained dynamic features based method. In this Chapter, I provide the detailed description of our integrated static and dynamic method. The experimental results prove our Hypothesis 4: Combining static and dynamic features can produce better detection and classification performance than any individual feature can produce.

The rest of the chapter is organized as follows: the relevant literature reviews are summarized in Section 8.2. In Section 8.3, I detail our analytical approach and data preparation and in Section 8.4 I describe the experimental set-up. Sections 8.5 present the integrated tests. In Section 8.7 I analyse and compare the results and summarize in Section 8.8.
8.2 Related Work

As I mentioned in Section 2.2 of Chapter 2, on the side of static analysis, researchers focused on static features, such as instructions, basic blocks, functions, control flow and data flow extracted from binary code. On the side of dynamic analysis, researchers focused on dynamic information extracted from the real executions of malware under a controlled environment.

In the following sections, I provide a detailed description of our integrated experiments.

8.3 Data Preparation

In the integrated experiments, we extract both static and dynamic features using the same methods mentioned in Chapter 4, 5 and 7. We use the same test dataset, Table 3.1 in Chapter 3, in this experiment. I recall how the vectors were generated in the next subsections.

8.3.1 FLF Vector

For extracting the frequency of function length we normalize the function length vector using the same procedure described in Section 4.5 of Chapter 4. As an example, considers an executable file with only 23 functions, with the functions having the following lengths (in bytes): 12, 12, 12, 12, 12, 12, 12, 50, 50, 50, 140, 340, 420, 448, 506, 548, 828, 848, 1342, 1344, 1538, 3138, 3580, 4072, 4632. For illustration purposes, let us create just 10 function length ranges, exponentially spaced. The distribution of lengths across the bins would be as shown in Figure 8.1. We then capture a vector of length 10 from the last column of this figure: (0, 0, 5, 3, 1, 3, 4, 5, 2, 0).
8.3.2 PSI Vector

Another static feature is PSI. As described in Chapter 5, we again use a vector representation, by first constructing a global list of all strings extracted from our ida2DBMS. The method is as described in Chapter 5 and is explained by means of the following example: Let us assume that the example global string list contains just 7 strings: \{"GetProcAddress", "RegQueryValueExW", "CreateFileW", "OpenFile", "FindFirstFileA", "FindNextFileA", "CopyMemory"\}. Now consider that the printable strings extracted from the executable file are as follows: "GetProcAddress", "RegQueryValueExW", "CreateFileW", "GetProcAddress". The PSI vector records the total number of distinct strings extracted and which of the strings in the global list are present. Figure 8.2 presents the corresponding data for this executable. The vector is then (3, 1, 1, 1, 0, 0, 0, 0).
8.3.3 Dynamic Vector

In the dynamic method, once again, a vector representation of the log data extracted after emulation is used in the classification. This process is described in detail in Chapter 7 and explained here in the following illustration. After running all the executable files in our sample set and logging the Windows API calls we extract the strings from the log files and again construct a global string list. The strings extracted include API function names and parameters passed to the functions. For the purpose of example, consider this global string list: \{ “RegOpenKeyEx”, “RegQueryValueExW”, “Compositing”, “RegOpenKeyExW”, “0x54”, “Control Panel\Desktop”, “LameButtonText”, “LoadLibraryW”, “.\UxTheme.dll”, “LoadLibraryExW”, “MessageBoxW” \}. Then for our example executable, we obtain the following abridged log file:

2010/09/02 11:24:50.217, 180, RegQueryValueExW, Compositing
2010/09/02 11:24:50.217, 180, RegOpenKeyExW, 0x54, Control Panel\Desktop
2010/09/02 11:24:50.217, 180, RegQueryValueExW, LameButtonText
2010/09/02 11:24:50.217, 180, LoadLibraryW, .\UxTheme.dll
2010/09/02 11:24:50.217, 180, LoadLibraryExW, .\UxTheme.dll
The strings extracted are highlighted. We then count the number of occurrences of each string in the global list. For this example, Figure 8.3 gives this data. The corresponding vector is then (0, 2, 1, 1, 1, 1, 1, 2, 1, 0).

<table>
<thead>
<tr>
<th>String</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;RegOpenKeyEx&quot; count</td>
<td>0</td>
</tr>
<tr>
<td>&quot;RegQueryValueExW&quot; count</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Compositing&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;RegOpenkeyExW&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;0x54&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Control Panle\Desktop&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;LameButtonText&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;LoadLibraryW&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;.\UxTheme.dll&quot; count</td>
<td>2</td>
</tr>
<tr>
<td>&quot;LoadLibraryExW&quot; count</td>
<td>1</td>
</tr>
<tr>
<td>&quot;MessageBoxW&quot; count</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 8.3. Example of Data Used in a Dynamic Feature Vector

8.3.4 Integrated Vector

Now I describe how we combine the feature vectors separately established in each of the above three methods into one vector. Figure 8.4 represents this process.

Our integrated vector contains features from all of the previous feature extraction methods, including FLF, PSI and Dynamic logs. The integrated vectors are constructed by concatenating the FLF, PSI and dynamic log vectors. Therefore for the above mentioned example executable, the integrated feature vector would be (0, 0, 5, 3, 1, 3, 4, 5, 2, 0, 3, 1, 1, 0, 0, 0, 0, 2, 1, 1, 1, 1, 1, 2, 1, 0) and this data can also be seen in Figure 8.5.
8.4 Experimental Set-up

8.4.1 Motivation

In Chapter 2 I discuss advantages and disadvantages of both static and dynamic analysis. We know that both of them have their merits and limitations.

Static analysis and extraction of executable files are the foundation of malware detection and classification, static analysis methods provide information about content and structure of a program and are well explored and widely adopted. They present a global view sight of analyzed executables, they have easily accessible forms, low level time and resource consuming. While at the same time, static analysis methods are susceptible to inaccuracies due to obfuscation and polymorphic techniques and fail to detect inter-component/system interaction information.

As with static analysis, dynamic analysis also has its merits and limitations. Dynamic analysis methods determine whether a program is malicious by observing the actual execution of it malicious behaviors, which is far more straightforward than
<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length 1-2 functions</td>
<td>0</td>
</tr>
<tr>
<td>Length 3-7 functions</td>
<td>0</td>
</tr>
<tr>
<td>Length 8-21 functions</td>
<td>5</td>
</tr>
<tr>
<td>Number of strings</td>
<td>3</td>
</tr>
<tr>
<td>&quot;GetProcAddress&quot; present</td>
<td>TRUE</td>
</tr>
<tr>
<td>&quot;RegQueryValueExW&quot; present</td>
<td>TRUE</td>
</tr>
<tr>
<td>&quot;CreateFileW&quot; present</td>
<td>TRUE</td>
</tr>
<tr>
<td>&quot;OpenFile&quot; present</td>
<td>FALSE</td>
</tr>
<tr>
<td>&quot;RegOpenKeyEx&quot; count</td>
<td>0</td>
</tr>
<tr>
<td>&quot;RegQueryValueExW&quot; count</td>
<td>2</td>
</tr>
<tr>
<td>&quot;LoadLibraryW&quot; count</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 8.5. Data Used in Generating an Integrated Feature Vector
just examining its binary code. The dynamic runtime information can also be directly used in assessing the potential damage malware can cause which enables detection and classification of new threats. While dynamic methods are trace dependence and have limited view of analyzed executables.

Static analysis alone is not enough to either detect or classify malicious code, likewise dynamic analysis is inadequate either. As I discussed in Chapter 7, due to the limitation of dynamic methodology, the overall performance of detection and classification dropped after we applied the dynamic method to the expanded dataset with more recent malware families, such as ‘Banker” and “Bancos”. This motivate us to combine static and dynamic method to complement each other to keep good level of performance.

In this chapter, I present our integrated experiments.

### 8.4.2 Test Dataset

In this experiment, we use the same test dataset in our dynamic experiment. See Table 3.1 in Chapter 3. There are 2939 executables, including Trojans, Worms, Viruses and Cleanware. The first 12 families have been pre-classified as Trojans; the next 2 families are Worms; and the last 3 families as Viruses.

### 8.5 Integrated Tests

After we generate the integrated feature vectors, we use the similar classification process described in Chapter 4, 5, 6 and 7. We input the integrated feature vectors into the WEKA classification system for which we have written an interface. As shown in Figure 8.6, 10-fold cross validation is used for classifying malware in the experiments.
In the previous chapters, I have already described this general classification process. To be clear, I restate this general classification process separately. In the cross validation procedure, we first select one family $F_i$ and divide it into groups $C_k$ of approximately equal size where $k$ varies from 1 to 10. Then we divide all the other executable files into subsets which have the same number of executable files as the selected one as far as possible. We then choose one subset and divide it into groups $Other_k$ of approximately equal size where $k$ varies from 1 to 10.

Next the classifier takes 9 groups from $C_k$ and $Other_k$ to set up the training set and the remaining one group from $C_k$ and $Other_k$ is used for the testing set. In our WEKA classification engine, we use a training set to build up the classification model and validate it by using the test set in order to obtain the statistical results. This process is performed once or repeated until all the subsets are traversed. The whole process is repeated for each family and we then calculate weighted average classification results.

Because of the significance to the anti-malware vendor of being able to satisfy customers that the anti-virus software will allow legitimate cleanware with executable characteristics to pass, testing to verify that malware can be distinguished from clean-
ware has become an important part of classification research [TIBV10]. Thus in this section I present two different tests: (i) malware families classification, and (ii) malware versus cleanware classification.

### 8.5.1 Family Classification

In this experiment, we test our classification system by comparing executable files across and within families. In order to do this family classification, we follow the process explained in Figure 8.6. To train the classifiers, we choose an equal number of positive (from a particular family) and negative (from other families) files. We construct the negative set by randomly selecting files from the other families. Table 8.1 and Table 8.2 show the family classification results.

### 8.5.2 Malware Versus Cleanware Classification

We now turn to the problem of distinguishing between cleanware files and malware files. In this experiment, we follow the recommendations of [MSF+08] in using an equal portion of malware and of cleanware in the testing and ensured that each file is incorporated into at least one test. We use 541 cleanware executable files and randomly selecting the same number of executable files from the malware set. Using this process, we generate 5 separate malware executable file groups, $MG_1, MG_2, \ldots, MG_5$, and test each group against a cleanware group. Referring to Figure 8.6, we use the cleanware group (particular family) as a positive set and select one group of malware (from $MG_i$, $i = 1, 2, \ldots, 5$) (other families) as a negative set. We then split the cleanware and malware group separately, as shown in Figure 8.6, for making training and test data. Table 8.3 and Table 8.4 show the weighted average of the experimental results.
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>Addclicker</td>
<td>65</td>
<td>0.3</td>
<td>0.266</td>
<td>71.6</td>
<td>0.25</td>
</tr>
<tr>
<td>Gamepass</td>
<td>179</td>
<td>0.194</td>
<td>0.1352</td>
<td>83.53</td>
<td>0.158</td>
</tr>
<tr>
<td>Banker</td>
<td>47</td>
<td>0.3</td>
<td>0.325</td>
<td>68.75</td>
<td>0.3</td>
</tr>
<tr>
<td>Frethog</td>
<td>174</td>
<td>0.058</td>
<td>0.0764</td>
<td>93.23</td>
<td>0.076</td>
</tr>
<tr>
<td>SillyAutorun</td>
<td>87</td>
<td>0.21</td>
<td>0.212</td>
<td>78.75</td>
<td>0.2</td>
</tr>
<tr>
<td>SillyDI</td>
<td>439</td>
<td>0.225</td>
<td>0.279</td>
<td>74.76</td>
<td>0.18</td>
</tr>
<tr>
<td>Vundo</td>
<td>80</td>
<td>0.05</td>
<td>0.087</td>
<td>93.125</td>
<td>0.1</td>
</tr>
<tr>
<td>Bancos</td>
<td>446</td>
<td>0.075</td>
<td>0.086</td>
<td>91.93</td>
<td>0.072</td>
</tr>
<tr>
<td>Clagger</td>
<td>44</td>
<td>0.02</td>
<td>0.03</td>
<td>97.5</td>
<td>0</td>
</tr>
<tr>
<td>agobot</td>
<td>283</td>
<td>0</td>
<td>0.007</td>
<td>99.64</td>
<td>0.003</td>
</tr>
<tr>
<td>ahideon</td>
<td>41</td>
<td>0</td>
<td>0.08</td>
<td>96.25</td>
<td>0.05</td>
</tr>
<tr>
<td>bambo</td>
<td>44</td>
<td>0</td>
<td>0.05</td>
<td>97.5</td>
<td>0</td>
</tr>
<tr>
<td>boxed</td>
<td>178</td>
<td>0</td>
<td>0.02</td>
<td>98.53</td>
<td>0.01</td>
</tr>
<tr>
<td>emerleox</td>
<td>75</td>
<td>0.01</td>
<td>0.02</td>
<td>97.85</td>
<td>0.01</td>
</tr>
<tr>
<td>looked</td>
<td>66</td>
<td>0</td>
<td>0.016</td>
<td>99.16</td>
<td>0</td>
</tr>
<tr>
<td>robknot</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>robzips</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Cleanfiles</td>
<td>541</td>
<td>0.03</td>
<td>0.02</td>
<td>97.22</td>
<td>0.04</td>
</tr>
<tr>
<td>Avg</td>
<td>2939</td>
<td>0.0818</td>
<td>0.0949</td>
<td>91.0736</td>
<td>0.0805</td>
</tr>
<tr>
<td>W. Avg</td>
<td>2939</td>
<td>0.0853</td>
<td>0.0959</td>
<td>90.8209</td>
<td>0.0801</td>
</tr>
</tbody>
</table>

Table 8.1. Integrated Test Result (Base Classifiers)
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addclicker</td>
<td>65</td>
<td>0.283</td>
<td>0.267</td>
<td>73.125</td>
<td>0.25</td>
</tr>
<tr>
<td>Gamepass</td>
<td>179</td>
<td>0.194</td>
<td>0.135</td>
<td>83.52</td>
<td>0.018</td>
</tr>
<tr>
<td>Banker</td>
<td>47</td>
<td>0.3</td>
<td>0.32</td>
<td>68.75</td>
<td>0.325</td>
</tr>
<tr>
<td>Frethog</td>
<td>174</td>
<td>0.07</td>
<td>0.076</td>
<td>93.38</td>
<td>0.076</td>
</tr>
<tr>
<td>SillyAutorun</td>
<td>87</td>
<td>0.21</td>
<td>0.212</td>
<td>78.75</td>
<td>0.22</td>
</tr>
<tr>
<td>SillyDI</td>
<td>439</td>
<td>0.204</td>
<td>0.258</td>
<td>83.18</td>
<td>0.179</td>
</tr>
<tr>
<td>Vundo</td>
<td>80</td>
<td>0.05</td>
<td>0.087</td>
<td>93.125</td>
<td>0.112</td>
</tr>
<tr>
<td>Bancos</td>
<td>446</td>
<td>0.09</td>
<td>0.08</td>
<td>94.025</td>
<td>0.08</td>
</tr>
<tr>
<td>Clagger</td>
<td>44</td>
<td>0.025</td>
<td>0.03</td>
<td>97.5</td>
<td>0</td>
</tr>
<tr>
<td>agobot</td>
<td>283</td>
<td>0</td>
<td>0.007</td>
<td>99.65</td>
<td>0.003</td>
</tr>
<tr>
<td>ahlrecon</td>
<td>41</td>
<td>0</td>
<td>0.08</td>
<td>96.25</td>
<td>0.05</td>
</tr>
<tr>
<td>bambo</td>
<td>44</td>
<td>0</td>
<td>0.05</td>
<td>97.5</td>
<td>0</td>
</tr>
<tr>
<td>boxed</td>
<td>178</td>
<td>0</td>
<td>0.02</td>
<td>98.52</td>
<td>0.002</td>
</tr>
<tr>
<td>emerleox</td>
<td>75</td>
<td>0.01</td>
<td>0.02</td>
<td>97.85</td>
<td>0.02</td>
</tr>
<tr>
<td>looked</td>
<td>66</td>
<td>0</td>
<td>0.016</td>
<td>99.16</td>
<td>0</td>
</tr>
<tr>
<td>robknot</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>robzips</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0.01</td>
</tr>
<tr>
<td>Cleanfiles</td>
<td>541</td>
<td>0.02</td>
<td>0.02</td>
<td>98.4</td>
<td>0.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Avg</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2939</td>
<td>0.0809</td>
<td>0.0932</td>
<td>91.8158</td>
<td>0.0775</td>
<td>0.0744</td>
<td>93.9331</td>
<td>0.0757</td>
<td>0.0962</td>
<td>94.3531</td>
</tr>
<tr>
<td>W. Avg</td>
<td>2939</td>
<td>0.0830</td>
<td>0.0918</td>
<td>92.6562</td>
<td>0.0758</td>
<td>0.0723</td>
<td>94.6701</td>
<td>0.0601</td>
<td>0.1006</td>
<td>95.678</td>
</tr>
</tbody>
</table>

Table 8.2. Integrated Test Result (Meta Classifiers)
<table>
<thead>
<tr>
<th></th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>Clean Vs Group1</td>
<td>0.011</td>
<td>0.035</td>
<td>97.68</td>
<td>0.007</td>
</tr>
<tr>
<td>Clean Vs Group2</td>
<td>0.027</td>
<td>0.037</td>
<td>96.75</td>
<td>0.035</td>
</tr>
<tr>
<td>Clean Vs Group3</td>
<td>0.02</td>
<td>0.05</td>
<td>96.23</td>
<td>0.029</td>
</tr>
<tr>
<td>Clean Vs Group4</td>
<td>0.01</td>
<td>0.04</td>
<td>97.41</td>
<td>0.007</td>
</tr>
<tr>
<td>Clean Vs Group5</td>
<td>0.025</td>
<td>0.057</td>
<td>95.74</td>
<td>0.034</td>
</tr>
<tr>
<td>Average</td>
<td>0.0186</td>
<td>0.0438</td>
<td>96.762</td>
<td>0.0224</td>
</tr>
</tbody>
</table>

Table 8.3. Base Classifiers on Malware Versus Cleanware Using Integrated Method
<table>
<thead>
<tr>
<th>Clean Vs Group</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>Clean Vs Group1</td>
<td>0.035</td>
<td>0.04</td>
<td>95.74</td>
<td>0.033</td>
</tr>
<tr>
<td>Clean Vs Group2</td>
<td>0.03</td>
<td>0.05</td>
<td>95.95</td>
<td>0.04</td>
</tr>
<tr>
<td>Clean Vs Group3</td>
<td>0.02</td>
<td>0.05</td>
<td>96.57</td>
<td>0.03</td>
</tr>
<tr>
<td>Clean Vs Group4</td>
<td>0.01</td>
<td>0.04</td>
<td>97.45</td>
<td>0.007</td>
</tr>
<tr>
<td>Clean Vs Group5</td>
<td>0.037</td>
<td>0.061</td>
<td>95.92</td>
<td>0.054</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.0264</td>
<td>0.0482</td>
<td>96.326</td>
<td>0.0328</td>
</tr>
</tbody>
</table>

Table 8.4. Meta Classifiers on Malware Versus Cleanware Using Integrated Method
8.5.3 Using the Integrated Method on Old and New Families

From Table 3.1 of Chapter 3, we can see that our experimental data set includes Trojans, Viruses and Worms spanning from 2003 to 2010. We divide them into two groups, the first collected between 2003 and 2008 including “Clagger”, “Robknot”, “Robzips”, “alureon”, “Bambo”, “Boxed”, “Emerleox”, “Looked”, “Agobot”, and the second collected between 2009 and 2010 including “Addclicker”, “Gamepass”, “Banker”, “Frethog”, “SillyAutorun”, “SillyDI”, “Vundo”, “Bancos”. To demonstrate that our method is robust to changes in malware development, we test and compare results on these two sets of malware in this experiments. Table 8.5 and Table 8.6 show the experimental results on old families and Table 8.7 and Table 8.8 show the experimental results on new families.

8.6 Running Times

In Table 8.9, running times, including ExportingTime and FLV GenerationTime, are listed for each family. All the times in this table are in seconds. As I mentioned in Section 4.7 of Chapter 4 and Section 5.6 of Chapter 5, time spent on exporting executables to ida2DBMS is the ExportingTime. And FLV GenerationTime is used to generate function length vectors by database programs described in Section 4.3.3 of Chapter 4.

Besides these preprocess times, the running times for the classification component of the experiments based on base classifiers are 110 mins for static FLF features, 368 mins for static PSI features, 348 mins for dynamic API call based features and 440 mins for the integrated test.
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>Clagger</td>
<td>44</td>
<td>0</td>
<td>0.05</td>
<td>97.5</td>
<td>0</td>
</tr>
<tr>
<td>Agobot</td>
<td>283</td>
<td>0.004</td>
<td>0.007</td>
<td>99.46</td>
<td>0.004</td>
</tr>
<tr>
<td>Alureon</td>
<td>41</td>
<td>0.025</td>
<td>0.05</td>
<td>96.25</td>
<td>0.075</td>
</tr>
<tr>
<td>Bambo</td>
<td>44</td>
<td>0</td>
<td>0.05</td>
<td>97.5</td>
<td>0</td>
</tr>
<tr>
<td>Boxed</td>
<td>178</td>
<td>0.006</td>
<td>0.029</td>
<td>98.23</td>
<td>0.006</td>
</tr>
<tr>
<td>Emerleox</td>
<td>75</td>
<td>0</td>
<td>0.014</td>
<td>99.28</td>
<td>0</td>
</tr>
<tr>
<td>Looked</td>
<td>66</td>
<td>0</td>
<td>0.016</td>
<td>99.16</td>
<td>0</td>
</tr>
<tr>
<td>Robknot</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Robzips</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.0039</td>
<td>0.024</td>
<td>98.5978</td>
<td>0.0094</td>
</tr>
<tr>
<td>W.Avg</td>
<td></td>
<td>0.0037</td>
<td>0.0178</td>
<td>98.9205</td>
<td>0.0059</td>
</tr>
</tbody>
</table>

Table 8.5. Weighted Average of Base Classifiers Results on Old Families Using Integrated Method
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clagger</td>
<td>44</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>Agobot</td>
<td>283</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Alureon</td>
<td>41</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>Bambo</td>
<td>44</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Emergeox</td>
<td>178</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>Looked</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Roblink</td>
<td>78</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Robzips</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td>75</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>W.Avg</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.6. Weighted Average of Meta Classifiers Results on Old Families Using Integrated Method
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
<th>Average</th>
<th>W. Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
</tr>
<tr>
<td>Addcicker</td>
<td>65</td>
<td>0.199</td>
<td>0.28</td>
<td>75.83</td>
<td>0.166</td>
<td>0.299</td>
<td>76.6</td>
</tr>
<tr>
<td>Gamepass</td>
<td>179</td>
<td>0.15</td>
<td>0.11</td>
<td>86.47</td>
<td>0.14</td>
<td>0.12</td>
<td>86.4</td>
</tr>
<tr>
<td>Banker</td>
<td>47</td>
<td>0.38</td>
<td>0.5</td>
<td>56.25</td>
<td>0.425</td>
<td>0.275</td>
<td>65</td>
</tr>
<tr>
<td>Frethog</td>
<td>174</td>
<td>0.05</td>
<td>0.094</td>
<td>92.64</td>
<td>0.064</td>
<td>0.071</td>
<td>93.23</td>
</tr>
<tr>
<td>SillyAutorun</td>
<td>87</td>
<td>0.212</td>
<td>0.212</td>
<td>78.75</td>
<td>0.212</td>
<td>0.175</td>
<td>80.62</td>
</tr>
<tr>
<td>SillyDI</td>
<td>439</td>
<td>0.23</td>
<td>0.28</td>
<td>74.41</td>
<td>0.16</td>
<td>0.176</td>
<td>82.67</td>
</tr>
<tr>
<td>Vundo</td>
<td>80</td>
<td>0.088</td>
<td>0.075</td>
<td>91.87</td>
<td>0.087</td>
<td>0.087</td>
<td>91.25</td>
</tr>
<tr>
<td>Bancos</td>
<td>446</td>
<td>0.107</td>
<td>0.125</td>
<td>88.4</td>
<td>0.107</td>
<td>0.093</td>
<td>90</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.177</td>
<td>0.2095</td>
<td>80.5775</td>
<td>0.1701</td>
<td>0.162</td>
<td>83.2213</td>
</tr>
<tr>
<td>W. Average</td>
<td></td>
<td>0.1586</td>
<td>0.1851</td>
<td>82.70496</td>
<td>0.1386</td>
<td>0.1365</td>
<td>86.0038</td>
</tr>
</tbody>
</table>

Table 8.7. Weighted Average of Base Classifiers Results on New Families Using Integrated Method
<table>
<thead>
<tr>
<th>Family</th>
<th>Num</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>Addclicker</td>
<td>65</td>
<td>0.199</td>
<td>0.283</td>
<td>75.83</td>
<td>0.216</td>
</tr>
<tr>
<td>Gamepass</td>
<td>179</td>
<td>0.15</td>
<td>0.11</td>
<td>86.5</td>
<td>0.12</td>
</tr>
<tr>
<td>Banker</td>
<td>47</td>
<td>0.375</td>
<td>0.5</td>
<td>56.25</td>
<td>0.4</td>
</tr>
<tr>
<td>Frethog</td>
<td>174</td>
<td>0.053</td>
<td>0.094</td>
<td>92.64</td>
<td>0.05</td>
</tr>
<tr>
<td>SillyAutorun</td>
<td>87</td>
<td>0.212</td>
<td>0.212</td>
<td>78.75</td>
<td>0.212</td>
</tr>
<tr>
<td>SillyDI</td>
<td>439</td>
<td>0.27</td>
<td>0.31</td>
<td>75.93</td>
<td>0.1</td>
</tr>
<tr>
<td>Vundo</td>
<td>80</td>
<td>0.0875</td>
<td>0.09</td>
<td>93.67</td>
<td>0.08</td>
</tr>
<tr>
<td>Bancos</td>
<td>446</td>
<td>0.14</td>
<td>0.12</td>
<td>90.68</td>
<td>0.1</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.1858</td>
<td>0.2149</td>
<td>81.2813</td>
<td>0.1596</td>
</tr>
<tr>
<td>W. Average</td>
<td></td>
<td>0.1799</td>
<td>0.1933</td>
<td>83.9136</td>
<td>0.1163</td>
</tr>
</tbody>
</table>

Table 8.8. Weighted Average of Meta Classifiers Results on New Families Using Integrated Method
<table>
<thead>
<tr>
<th>Family</th>
<th>No. Samples</th>
<th>Exporting Time (secs)</th>
<th>FLV Generation Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bambo</td>
<td>44</td>
<td>779</td>
<td>140</td>
</tr>
<tr>
<td>Boxed</td>
<td>178</td>
<td>9832</td>
<td>9269</td>
</tr>
<tr>
<td>Alureon</td>
<td>41</td>
<td>1865</td>
<td>607</td>
</tr>
<tr>
<td>Robknot</td>
<td>78</td>
<td>2010</td>
<td>100</td>
</tr>
<tr>
<td>Clagger</td>
<td>44</td>
<td>459</td>
<td>22</td>
</tr>
<tr>
<td>Robzips</td>
<td>72</td>
<td>2356</td>
<td>533</td>
</tr>
<tr>
<td>SillyDl</td>
<td>439</td>
<td>11049</td>
<td>12754</td>
</tr>
<tr>
<td>Vundo</td>
<td>80</td>
<td>1728</td>
<td>2324</td>
</tr>
<tr>
<td>Gamepass</td>
<td>179</td>
<td>4345</td>
<td>5201</td>
</tr>
<tr>
<td>Bancos</td>
<td>446</td>
<td>12573</td>
<td>12958</td>
</tr>
<tr>
<td>adclicker</td>
<td>65</td>
<td>2039</td>
<td>1888</td>
</tr>
<tr>
<td>Banker</td>
<td>47</td>
<td>3318</td>
<td>1366</td>
</tr>
<tr>
<td>Frethog</td>
<td>174</td>
<td>1044</td>
<td>5055</td>
</tr>
<tr>
<td>SillyAutorun</td>
<td>87</td>
<td>1635</td>
<td>2528</td>
</tr>
<tr>
<td>Agobot</td>
<td>283</td>
<td>40203</td>
<td>33401</td>
</tr>
<tr>
<td>Looked</td>
<td>66</td>
<td>4001</td>
<td>1577</td>
</tr>
<tr>
<td>Emerleox</td>
<td>75</td>
<td>7031</td>
<td>519</td>
</tr>
<tr>
<td>Cleanware</td>
<td>541</td>
<td>70165</td>
<td>15718</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2939</strong></td>
<td><strong>176432</strong></td>
<td><strong>105960</strong></td>
</tr>
</tbody>
</table>

Table 8.9. Running Times in the Integrated Experiments

### 8.7 Discussion

#### 8.7.1 Family Classification

Table 8.10 compares the results of old and integrated methods based on meta classifiers. Figures 8.7, 8.8, 8.9 illustrate that integrated method outperforms both static and dynamic methods. From these figures we see that integrated method always presents lowest false positive rate, lowest false negative rate and highest classification accuracy. Figure 8.9 shows that our four methods are clearly separated in classification accuracy, with the FLF method showing poorest performance among the presented methods, followed by PSI, then dynamic and finally the integrated method achieving the best results. Once again, the Meta-RF classifier achieved the best overall results in family classification.
<table>
<thead>
<tr>
<th>Methods</th>
<th>SVM</th>
<th>IB1</th>
<th>DT</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
<td>FP</td>
</tr>
<tr>
<td>FLF</td>
<td>0.2977</td>
<td>0.121</td>
<td>79.057</td>
<td>0.1728</td>
</tr>
<tr>
<td>PSI</td>
<td>0.1401</td>
<td>0.1729</td>
<td>84.347</td>
<td>0.1452</td>
</tr>
<tr>
<td>Dynamic</td>
<td>0.0978</td>
<td>0.1051</td>
<td>89.8495</td>
<td>0.1271</td>
</tr>
<tr>
<td>Integrated</td>
<td>0.0830</td>
<td>0.0918</td>
<td>92.6562</td>
<td>0.0758</td>
</tr>
</tbody>
</table>

Table 8.10. Comparison of Old and Integrated Methods (Meta-classifiers).
Figure 8.7. Compare FP Rate of Old and Integrated Methods (Meta-classifier)

Figure 8.8. Compare FN of Old and Integrated Methods (Meta-classifier)
Figure 8.9. Compare Accuracy of Old and Integrated Methods (Meta-classifier)

### 8.7.2 Malware Versus Cleanware Classification

In Table 8.11, we compare our results with similar work classifying malware versus cleanware. Unless stated otherwise, the results given are the best accuracy obtained. In Santos et al., [SNB11], the authors used static byte n-gram features for their classification using semi-supervised learning algorithms. They used 2,000 executable files in their experiment and obtained 88.3% accuracy. In Ye et al., [YLCJ10], the authors used static API call sequences as the characterizing feature with 30,601 executable files and obtained 93.69% accuracy. Wang et al., [WPZL09], used static API call sequences distilled from 714 PE files and pre-identified as normal (361) and virus programs (353); they achieved 93.71% classification accuracy. In Moskovitch et al., [MSF+08], the authors used the text categorization process and examined the relationship between the malicious file percentage (MFP) in the test set. They found that the best mean performance is associated with a 50% MFP in the training set, which is 95%. Z. Shafiq [STF09] compares the performance of evolutionary and non-evolutionary rule learning algorithms for classifying between malware and cleanware. Their reported best average accuracy is 99.75% (SLIPPER, a non-evolutionary rule based algorithm). In our earlier work, [TIBV10], we used dynamic API calls and
their parameters as features and obtained 97.3% accuracy based on the classification method used there, while our current integrated method achieved 97.46% accuracy which is best among the other works.

According to the Table 8.11, the best accuracy reported in [STF09]. However, in their paper they have statically extracted 189 attributes from executable files and used this as the basis for classification. Our method uses both static and dynamically extracted features. Our results show that combining static and dynamic features greatly improves classification accuracy. Shafiq [STF09] considers a broader range of static features than described in this paper. The evidence in this paper predicts that combining this with a similarly rich set of dynamic features would further improve classification accuracy and robustness.

8.7.3 Using the Integrated Method on Old and New Families

As I mentioned above, to test if our method is robust to changes in malware development, we also do experiments on two separated sets: old families and new families. Table 8.12 compares the weighted average of old family data, new family data and combined data using integrated method. And Figures 8.10, 8.11, 8.12 illustrate these comparison.

The results shows that the age (as measured by when the executable file was first collected) of the malware used has an impact on the test results. It is clear from the figures that the classifiers perform better on the old families. Since our classification method is less effective on the latest malware executables as compared to the older files, this demonstrates that malware continues to evolve and to deploy advanced anti-detection techniques.

From Table 8.6 we can see that the best weighted average accuracy of inte-
<table>
<thead>
<tr>
<th>Citations</th>
<th>Experimental data size</th>
<th>Feature extraction method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Santos et al. (2011) [SNB11]</td>
<td>2000</td>
<td>OpCode sequences</td>
<td>about 88.3%</td>
</tr>
<tr>
<td>Ye et al. (2010) [YLCJ10]</td>
<td>50000</td>
<td>PE instructions</td>
<td>93.69%</td>
</tr>
<tr>
<td>Wang et al. (2009) [WPZL09]</td>
<td>714</td>
<td>API call sequence</td>
<td>93.71%</td>
</tr>
<tr>
<td>Moskovitch et al. (2008) [MSF+08]</td>
<td>30601</td>
<td>byte n-grams</td>
<td>95%</td>
</tr>
<tr>
<td>Z. Shafiq et al. (2009) [STF09]</td>
<td>11786</td>
<td>clean with malicious files</td>
<td>99.75%</td>
</tr>
<tr>
<td>Sathyanarayan et al. (2008) [SKB08]</td>
<td>126</td>
<td>API calls</td>
<td>Did not mention the quantitative accuracy</td>
</tr>
<tr>
<td>R. Tian et al. (2010) [TIBV10]</td>
<td>1823</td>
<td>API calls and API parameters</td>
<td>97.30%</td>
</tr>
<tr>
<td>Our method</td>
<td>2939</td>
<td>Integrated (FLF, PSI, API calls and API parameters)</td>
<td>97.46%</td>
</tr>
</tbody>
</table>

Table 8.11. Comparison of Our Integrated Method with Similar Methods Based on Malware Versus Cleanware Testing.
grated method tested on old families is 99.8%, and three family, including “Emer-leox”, “Robknot” and “Robzips” come to 100%.

The weighted average results on new families are not as good as old families. From Table 8.8 we can see that Banker family gives the poorest performance. As I discussed in Section 7.6 of Chapter 7, due to the disadvantages of dynamic method, “Banker”, “Bancos” and “Adclicker” families give poor performance in dynamic method according to accuracy. From Table 7.7 of Chapter 7, we can see that in the dynamic test the best classification accuracy of “Banker”, “Bancos” and “Adclicker” families are 68.75%, 88% and 76.67% accuracy respectively.

We can see that when a mix of old and new malware is used, the overall performance is improved. The improvement is almost 9% with meta-SVM. This indicates the importance of including both old and new malware when developing new techniques: the integrated test performed better on the combined data set than on the new malware alone, so classification of new malware should be done with old malware also present in the data set.

Although the weighted average results on new families are not as good as old families, they still achieve best classification accuracy of 94.4%. Furthermore in Table 8.12, we achieve best weighted average accuracy of 97.1% using the integrated method tested on the combined data set, which proves again that our method is robust to changes in malware evolution.
<table>
<thead>
<tr>
<th>Meta Classifier</th>
<th>Old Family Data</th>
<th>New Family Data</th>
<th>Combined Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP</td>
<td>FN</td>
<td>Acc</td>
</tr>
<tr>
<td>SVM</td>
<td>0.003</td>
<td>0.017</td>
<td>98.924</td>
</tr>
<tr>
<td>IB1</td>
<td>0.005</td>
<td>0.015</td>
<td>99.168</td>
</tr>
<tr>
<td>DT</td>
<td>0.004</td>
<td>0.017</td>
<td>99.155</td>
</tr>
<tr>
<td>RF</td>
<td>0.002</td>
<td>0.007</td>
<td>99.82</td>
</tr>
</tbody>
</table>

Table 8.12. Comparison of Weighted Average of Old and New Malware Using Integrated Method.

![False Positive Rate](image1)

Figure 8.10. Compare FPRate of Old and New Malware Families Using Integrated Method.

![False Negative Rate](image2)

Figure 8.11. Compare FNRate of Old and New Malware Families Using Integrated Method.
8.7.4 Performance Analysis of Integrated Method

8.7.4.1 Effectiveness and Efficiency

In Table 8.10, I compare the results of our static, dynamic and integrated methods. Integrated method outperforms both static and dynamic methods. In addition, the integrated method is more efficient than using the other three methods combined. As I presented above, the running time for the classification component of the integrated test with base classifiers is 440 minutes. This compares favourably with a combined time of 826 minutes for running the three tests separately. However, the three tests could be run in parallel with a time just slightly higher than that of the slowest test which is 368 minutes. As long as the accuracy of the integrated test is at least as good as any of the accuracies in the other three tests, we have a strong argument for the integrated approach.

8.7.4.2 Robustness of Integrated System

Table 8.13 gives an analysis of robustness of our integrated system. From this table, we can conclude that:
• FLF (Function Length Frequency) based static method is effective when we applied it to a relatively small set of malware which only contains trojans. In that experiment, we achieved 92.63% weighted average malware detection and classification accuracy.

• PSI (Printable String Information) based static method is also effective when we applied it to a set of malware by introducing viruses into our original test dataset. In that experiment, we achieved 97.5% weighted average malware detection and classification accuracy.

• Next when we combined these two static methods and applied to the same test dataset as PSI method, we achieved an improved 98.89% malware detection and classification accuracy. This verified our Hypothesis 2: Combining several static features can produce better detection and classification performance than any individual feature can produce.

• We furthermore extended our test dataset by introducing more recent new malware families which are collected between 2009 and 2010. In addition, we introduced worms collected between 2009 and 2010 into our test dataset. Then we applied our dynamic method to this extended test dataset. Although the classification accuracy dropped to 90.4% due to the limitation of dynamic methodology, the experimental results showed that our dynamic method is still effective when tested on malware collected over an extended period of time.

• To make it comparable, we did FLF and PSI tests on the extended test dataset and achieved 87.82% and 87.81% accuracy respectively. Finally we integrated static and dynamic methods into a single test which gave us an significantly improved performance with 97.06% malware detection and classification accuracy.
<table>
<thead>
<tr>
<th>Chapters</th>
<th>Malware Type</th>
<th>Test Dataset Size</th>
<th>Method</th>
<th>Weighted Average Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Trojan</td>
<td>7 families and 721 files</td>
<td>FLF</td>
<td>0.9263</td>
</tr>
<tr>
<td>5</td>
<td>Trojan and Virus</td>
<td>10 families and 1367 files</td>
<td>PSI</td>
<td>0.975</td>
</tr>
<tr>
<td>6</td>
<td>Trojan and Virus</td>
<td>10 families and 1367 files</td>
<td>FLF and PSI</td>
<td>0.9886</td>
</tr>
<tr>
<td>7</td>
<td>Trojan, Worm and Virus</td>
<td>17 families and 2398 files</td>
<td>Dynamic</td>
<td>0.904</td>
</tr>
<tr>
<td>8</td>
<td>Trojan, Worm and Virus</td>
<td>17 families and 2398 files</td>
<td>FLF</td>
<td>0.8782</td>
</tr>
<tr>
<td>8</td>
<td>Trojan, Worm and Virus</td>
<td>17 families and 2398 files</td>
<td>PSI</td>
<td>0.8781</td>
</tr>
<tr>
<td>8</td>
<td>Trojan, Worm and Virus</td>
<td>17 families and 2398 files</td>
<td>Integrated</td>
<td>0.9706</td>
</tr>
</tbody>
</table>

Table 8.13. Robustness Analysis
8.8 Summary

In this Chapter, I gave a detailed account of our integrated method. First I presented the data preparation, following this I described our Experimental Set-up by explaining the motivation of our experiment and the dataset used in the tests. Then I elaborated our three integrated experiments, including Family Classification, Malware Versus Cleanware Classification and Using the Integrated Method on Old and New Families. Finally I analysed and discussed the performance of our system based on the results from these experiments. From these results and analysis, we can see that our integrated system is a scalable and robust malware detection and classification system.
Chapter 9

Conclusions

9.1 Accomplishments

In this thesis, the following hypotheses have been proposed and verified:

- **Hypothesis 1**: It is possible to find static features which are effective in malware detection and classification. From the experimental results in Chapter 4 and 5, we can see that both function length based features and printable string based features extracted by our static methods are effective in malware detection and classification. We achieved an average detection and classification accuracy of 87.7% and 97.5% respectively.

- **Hypothesis 2**: Combining several static features can produce better detection and classification performance than any individual feature can produce. Experimental results in Chapter 6 showed that combining the FLF and PSI features improved the performance by achieving the best weighted average accuracy of 98.9%.

- **Hypothesis 3**: It is possible to find dynamic features which are effective in
malware detection and classification. This hypothesis was verified by the experimental results in Chapter 7. In that experiment, we achieved the best weighted accuracy of 90.4% for family classification, and the best weighted average accuracy of 95.15% for malware versus cleanware tests.

- **Hypothesis 4**: Combining static and dynamic features can produce better detection and classification performance than any individual feature can produce. Experimental results from our integrated experiments in Chapter 8 indicated that combining the static and dynamic features method outperformed any individual method for all four classifiers. When we applied all the methods to an expanded test dataset by adding more recent malware in the integrated experiment, FLF, PSI and the Dynamic method achieved the best accuracy of 87.8%, 87.8% and 80.4% respectively, while the integrated method achieved the best accuracy of 97.1%.

- **Hypothesis 5**: Good levels of malware detection and classification accuracy can be retained on malware collected over an extended period of time. Our integrated experiments also indicated that the detection and classification accuracy is still maintained when applied to an extended test dataset with more recent malware samples.

Antivirus research is a relatively new research field with some work are still performed manually, there is a lack of structural analysis of a malware detection and classification system. In developing this thesis, some research questions and corresponding solutions were formed gradually. Based on these questions and solutions, I have proposed the architecture for a malware detection and classification system and presented its implementation. We have provided the antivirus community with answers for the following questions, which have been identified as also being valuable for their work.
1 *Data Collection and Data Preprocess*: Where do we collect test samples? How do we preprocess these samples to make them well-formed and fitting for the research?

2 *Data Storage*: How do we store and maintain this data in a reliable and safe way?

3 *Extraction and Representation*: What information should we extract from the executable files and how should we abstractly represent it based on the extraction.

4 *Classification Process*: Selection of suitable classification algorithms which need to be generalized based on the extracted feature sets.

5 *Performance Assessment*: How to do statistical detection of classification results and how to evaluate the performance of a system.

Regarding question 1 In Section 3.3.1 of Chapter 3, We have investigated reports from several antivirus companies regarding recent security threats. Based on the analysis of these reports, we focused our experiments on Trojans, Worms and Viruses. In addition, our project was supported by CA Technologies which has a long history of research and development in anti-virus products. To make the best use of this advantage, we collected three types of malware, including Trojans, Worms and Viruses from CA’s VET zoo, which have been pre-classified using generally acceptable mechanical means. We also collected clean executables from Window platforms spanning Windows 98 to Windows XP.

The original executable files that we collected were stored in the file system in the form of binary code. We needed to preprocess these files to make them suitable for our research. We used two methods of data preprocessing in our system as discussed
in Section 3.3.2 and Section 3.3.3 of Chapter 3. These methods included static and dynamic methods. In the static method, we first unpacked malware then performed a reverse engineering analysis of the executable files by IDA Pro and exported disassembling information into our ida2DBMS schema by using the customized software Ida2sql. In the dynamic method, the executable files were executed under a controlled environment which was based on Virtual Machine Technology. A trace tool was developed to monitor and trace the execution of malware. After execution, a log file was generated for each executable file, which became the base of our dynamic analysis.

**Regarding question 2** Considering accessibility, manoeuvrability and security of data, we stored and maintained our data in two effective and safe ways, including a database schema in Section 3.4.1 and log files in Section 3.4.2 of Chapter 3. In the static method, we chose DBMS as our data storage system because DBMS has many benefits which can facilitate our work, such as an integrated management interface, with data fetched in a simple and effective way plus data that is independent and safe. In the dynamic method, we stored dynamic information in the log files which wrote down the intercepted windows APIs.

**Regarding question 3** Two kinds of features were extracted from the executables, including the static features and dynamic features. Our aim was to analyze and extract simple and effective features from executable files. In the static method, we chose FLF in Chapter 4, and PSI in Chapter 5 as our static features. In the dynamic method, we chose intercepted windows APIs as our dynamic features in Chapter 7. Because our proposed system aimed to build a robust system which integrated the dynamic analysis and static analysis approaches, we combined two
kinds of static features mentioned above in Chapter 6 and integrated both static and dynamic features in Chapter 8.

**Regarding question 4** In Section 3.6.2 of Chapter 3, we looked into five kinds of Machine Learning classification algorithms. There were NB (Naïve Bayesian classifiers), IB1 (Instance-Based Learning), DT (Decision Tree), RF (Random Forest), and SVM (Support Vector Machine). Based on the understanding of their principles, analysis of work from other researchers, the advantages of these algorithms and their excellent performance in classification tasks, we applied these five algorithms along with AdaBoost in our experiments.

**Regarding question 5** We adopted the *k*-fold Cross Validation method in the training and testing phases of classification processes. To assess the classification results, some measures were introduced, including \( TPrate \), \( FPrate \), \( FNrate \), \( Precision \) and \( Accuracy \) in Section 3.7 of Chapter 3.

### 9.2 Weaknesses of methodologies

The work presented in this thesis still in its research phase, the following aspects need to be improved when it is put into industry practice:

**Respond in a timely manner** In anti-virus industry, timely response is critical. Detection of malware and malware family classification should be completed instantaneously. My current research work focus is on improving the classification accuracy. I integrated both static and dynamic features into a broader feature vector to improve the classification accuracy. Due to the fact that the feature set expands, the response time will increase. In addition, compared with huge number of malware released ev-
Every day, the test dataset in this thesis is small. I need to introduce more malware into the system. That means the size of features set may increase correspondingly, which would further affect the response time of the method. In the future work, I will introduce feature reduction or selection technology to minimize this influence.

**Static Methodologies** FLF(function length frequency) and PSI(printable string information) are the main features in static methodologies. FLF features depend on function length information extracted from executables by IDA Pro. And PSI features are extracted from Strings window of IDA Pro. These static methodologies are vulnerable to obfuscation and polymorphic techniques. Although these two features can complement each other, if malware writers obfuscate both of them, the detection and classification accuracy will be affected. To deal with this, in the future work, some other significant static features would be investigated and introduced into our system.

**Dynamic methodology** In dynamic methodology, we create virtual machine with networking disabled and run each executable for 30 seconds, in this case, analysis results are only based on malware behavior during one specific execution run and some of malware’s behavior can not be monitored due to the lack of execution conditions. In addition, in the dynamic method described in this thesis, trace tool name “HookMe” is used to monitor and trace the real execution of each file. Some popular and critical API call are monitored during the execution, malware writers can evade this detection by changing API calls invoked in the malware. To deal with this, in the future work, an updated API calls list should be introduced into the system.
9.3 Further Research Directions

Malware Data Set  In Section 1.1.3 of Chapter 1, I described the general types of malware, including Worms, Viruses, Trojans, Rootkits, Backdoors, Botnets, Hacker Utilities and other malicious programs. At this point in time, our research has focused on Trojans, Worms and Viruses. With the development and evolution of malware, we simply can not group them into the categories mentioned above, with malicious code combining two or more categories that can lead to powerful attacks. For instance, a worm containing a payload can install a backdoor to allow remote access.

In future work, we will introduce more types of malware and more recent malware into our system.

Features Extraction  As we know from the ida2DBMS schema, besides function length, and printable strings, we also store many other disassembled information, including basic blocks, instructions, control flow graph, call graph and some statistical information. In our future work, we can investigate this information to excavate more significant static features. Using the current dynamic method, we generated the feature vectors using frequency of intercepted windows APIs. While we think the sequences or subsequences of intercepted windows APIs contain scheduled actions taken by malware during its execution, these scheduled actions actually imply the significant behavioral information of malware. In future work, these sequences can be taken into consideration when we generate dynamic features by examining the log files.

Feature Reduction  In future work, we will introduce feature reduction or selection technology into our system. Feature reduction is the technology which reduces features to improve the effectiveness of the classification system by selecting a subset
of important relevant features upon which to focus its attention, while ignoring the rest. During feature reduction, the system probes the analysed data to acquire a better overall understanding about the data, ascertaining inter-relationships of features and data, and determining the key or important features.

Feature selection techniques can be categorized according to a number of criteria. One popular categorization are the terms “filter” and “wrapper” to describe the nature of the metric used to evaluate the worth of features [KJ97, HH03]. Wrappers evaluate each potential subset of features by using the classification accuracy provided by the actual target learning algorithm. Filter methods rely on tracing general properties of features to evaluate them and operate independently of any learning algorithm. We have already started to apply some filter selection methods into our system by implementing the experiments. In this thesis, I have not presented these due to time and space limitations.

**Classification Algorithms** The next aim is to optimize our system. The basic purpose of genetic algorithms is optimization [BMB93, KCS06]. So in future work, we will study genetic algorithms and integrate them into our classification system.
Bibliography


206


A brief history of malware. white paper, McAfee System Protection Solutions, October 2005.


[Vuk] Miha Vuk. Roc curve, lift chart and calibration plot.


214
Appendix A

Function Distance Experiment

A.1 Motivation For Experiment

Malware is essentially a piece of program code which without exception should follow the principle of computer programming techniques. Whatever a program is complicated, it is always composed of all kinds of self-contained software routines, functional units, which perform a certain task as defined by a programmer and may be considered logically as subject investigated when people analyze a program. As I mentioned in Chapter 4.1, I discussed about the reason that we started our research work from IDA functions.

In this part, I present another function distance based experiment. In this experiment, I apply three kinds of distance algorithms to the binary code of different functions to get the similarity between them, and based on the distance values, the similarity between two files is calculated.

In Section A.2, I recite function fetch process used in this experiment. In Section A.3, I describe three kinds of distance algorithms applied in this experiment. Section A.4 presents the experiment and Section A.5 gives the experimental results. In Section A.6 analyzes the performance of this experiment based on experimental results.
A.2 Function Fetch

From figure A.1, we can see that an IDA function is composed of many basic blocks; and each basic block is composed of instructions. So in my methodology, all the instructions and all the basic blocks that belong to a function are traversed and put them together to form the data of that function. For more detail information of functions, basic blocks and instructions, please refer to appendix D.

Figure A.1. Related Tables

For each function in the disassembling module of a specific executable file, I first get all the basic blocks belonging to that function; and then identify and fetch all the instructions of each basic block by using the value of basic_block_id. Then combine these instructions to form a hex format string for each basic block, in the same way, combine all the basic blocks belonging to the function to form a hex format string representing the function. Please refer to Figure 4.4 in Chapter 4 for the
detailed flow of this process. When I calculate the similarity between two functions, these hex format strings are used as the parameters of each distance algorithms. Now I would like to introduce three distance algorithms that I used in this experiment.

A.3 Distance Algorithms

I use three distance algorithms, they are Levenshtein distance (LD), q-gram and LLCS (length longest common subsequence).

A.3.1 LD Distance Algorithm

Levenshtein distance (LD) is a measure of the similarity between two strings. The distance is the number of deletions, insertions, or substitutions required to transform a string into another. The greater the Levenshtein distance, the more different the strings are. For example:

- String1="'this is an apple'"
- String2="'this is an apple'"
- String3="'this is an orange"

Then the distance between two strings are:

- LD(String1,String2) = 0, because no transformations are needed. The strings are already identical.
- LD(String1,String3) = 5, because the minimum number of operations needed to transform String1 into String3 is 5, that is using "'oran'" replace "'appl'" and then insert "'g'".

The Levenshtein distance algorithm has been used in the following research areas:

- Spell checking
• Speech recognition
• DNA analysis
• Plagiarism detection

A.3.2 q-gram Distance Algorithm

q-gram is used in approximate string matching by "sliding" a window of length q over the characters of a string to create a number of length grams for matching a match is then rated as number of q-gram matches within the second string over possible q-grams. Here I use \( q = 3 \). For example, here are two string "dabc" and "abcde". The positional q-gram of the first string "dabc" are 
\[(1,##d),(2,#da),(3,dab),(4,abc),(5,bc#),(6,c##).\]
The positional q-gram of the second string "abcde" are 
\[(1,##a),(2,#ab),(3,abc),(4,bcd),(5,cde),(6,de#),(7,e##).\]
Here "##" indicates the beginning and end of the string.

So getting the q-grams for two strings allows the count of identical q-grams over the total q-grams available. As for the above example, the number of total q-grams is 13, and the identical q-grams are (4,abc) in first string and (3,abc) in second string, so the similarity based on q-gram distance of these two strings 2/13.

Q-gram is needed in many application areas, such as bio-computing, recognition of natural language and approximate string processing in DBMS.

A.3.3 LLCS Distance Algorithm

LLCS (length longest common subsequence). Given two sequences of characters, LLCS distance is the length of the longest common subsequence of both sequences. For example:

• String1="this is an apple"
• String2= "this is an apple"
The LLCS distance is 13, the longest subsequence is "‘this is an ae’". There are several applications of LLCS. Such as Molecular biology, File comparison and Screen redisplay etc.

A.4 Experimental Set-up

A.4.1 Experimental Data

The malware analysed in this experiment is from CA’s VET zoo (www.ca.com); thus it has been pre-classified using generally acceptable mechanical means. I use the similar method mentioned in Chapter 3.3.2.1 to unpack them and export them into our ida2DBMS schema. This experiment is test on a small set of malware files, which are 113 malware files from 11 families and they are all Trojans.

<table>
<thead>
<tr>
<th>Family</th>
<th>Number of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clagger</td>
<td>18</td>
</tr>
<tr>
<td>Cuebot</td>
<td>14</td>
</tr>
<tr>
<td>Ditul</td>
<td>4</td>
</tr>
<tr>
<td>Duiskbot</td>
<td>12</td>
</tr>
<tr>
<td>Robknot</td>
<td>18</td>
</tr>
<tr>
<td>Robzips</td>
<td>17</td>
</tr>
<tr>
<td>Abox</td>
<td>5</td>
</tr>
<tr>
<td>AdClicker</td>
<td>10</td>
</tr>
<tr>
<td>alemod</td>
<td>5</td>
</tr>
<tr>
<td>aliseru</td>
<td>5</td>
</tr>
<tr>
<td>allsum</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>113</strong></td>
</tr>
</tbody>
</table>

Table A.1. Number of Samples Used in FDE Experiment

A.4.2 Function Distance Experiment

The initial idea is to calculate similarity between two malware executable files. We need to calculate the distance or similarity between any two functions from these two files and based on these similarity, the similarity between two malware executable
files is calculated. Let $A = \{F_1, F_2, \ldots, F_M\}$ represent malware file $A$ which has $M$ functions. And let $B = \{f_1, f_2, \ldots, f_N\}$ represent malware file $B$ which has $N$ functions. We calculate all the similarity between any two functions from file $A$ and file $B$. Based on these similarities, a matrix is constructed between two files. To be clarify, we assumed file $A$ has 5 functions and $B$ has 4 functions. The following is an example of this similarity matrix:

$$SIM_{A,B} = \begin{pmatrix}
0.75 & 0.85 & 0.91 & 0.6 \\
0.82 & 0.79 & 0.83 & 0.75 \\
0.9 & 0.93 & 0.87 & 0.6 \\
0.78 & 0.8 & 0.9 & 0.65 \\
0.86 & 0.9 & 0.45 & 0.55
\end{pmatrix}$$

As for each row and column, we seek the element that has the maximum similarity values, we define them as $MaxR_1, MaxR_2, \ldots, MaxR_M$ for each row and $MaxC_1, MaxC_2, \ldots, MaxC_N$ for each column. For example, in the above example, the value of $MaxR_2$ is 0.83 and $MaxC_3$ is 0.91. In each row and column of that matrix, the element that has the maximum value could be the most possible matched function pair in that row or column. I use a threshold $T$ to determine if it is the case. In the experiment, I chose the value of $T$ as 0.75, 0.8, 0.85. We then count the number of most possible match function for row and column. Let $R_i$ indicated if the maximum value of $i$th row is greater than the threshold $T$ and $C_j$ indicates if the maximum value of $j$th column is greater than threshold $T$. If these maximum values are greater than the threshold $T$, we count it, otherwise, we just ignore them. In such a way, we calculate similarity between two files by the following formula:

$$S_{A,B} = \frac{\sum_{i=1}^{M} R_i/M + \sum_{j=1}^{N} C_j/N}{2}$$

(A.4.1)

where

$$R_i = \begin{cases}
1 & \text{if } MaxR_i \geq T \\
0 & \text{otherwise}
\end{cases}$$
\[ C_j = \begin{cases} 
1 & \text{if MaxC}_j \geq T \\
0 & \text{otherwise}
\end{cases} \]

In the above example, if we choose \( T = 0.85 \) the similarity between two file \( A \) and \( B \) is 0.78. For each malware file, this procedure is repeated to get the similarity between this malware and all the others.

### A.5 Experiment Results

After we get the similarity between two malware files by using the above method, I then visualize the result to show the similarity within the same family and across different families. The following figure A.2 is comparison of similarity of one malware file from clagger family with all the other malware files.

![Figure A.2. Experment Results](image)

In the figure A.2, the similarity values of the first group are the similarity between
a clagger malware file and all the other malware files from clagger family, and the rest groups are from all the other 10 families and each group represents a family. We can see that the similarity values among clagger family are relatively higher than the similarity values between this clagger malware file and other files from other families. And in some cases the similarity values within a family are very close, for instance, the majority of similarity values within in Robknot and Robzips are quite close.

From the figure, we can see that malware files from same family are quite similar compared to the similarity between two malware files from different families even we do this test on a relatively small dataset.

### A.6 Performance Analysis

This clustering method requires the computation of the distances between all pairs of functions and furthermore between all pairs of malware files, which invariably results in a computational complexity of $O(n^2)$. To evaluate the effectiveness of this method, I do another test that assesses the application of three algorithms to our data. I randomly create two hex format strings, that means every character of the string is valued between '0' – '9' and 'a' – 'f', and feed them into the distance algorithms. I record the time before every algorithm and after it, by this way, the execute time which measured in millisecond is acquired. Then change the length of these two strings to do the same test. Based on the test set in our ida2DBMS, the minimum length of the function in the database is 2 and the maximum length is 23782. So in this test the length of the two strings varies from the following values: 50, 100, 500, 1000, 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000. And this emulation is tested in a computer with Intel(R) Core(TM) CPU, 1.99GB RAM and windows XP operating system and JVM 1.6.0.03.

The following table gives the result of this test:

In table A.2, I list the time consumed by computing the n-gram distance value between two string, $s_1$ and $s_2$ and execution time is recorded in millisecond. Here the sign $X$ means the current computing environment can not handle it, that is it exceed the limit of JVM (Java Virtual Machine) heap space, the algorithm will halt.
Table A.2. Execution Results

![Figure A.3. Execution Time Trend](image_url)
by throwing an exception. From this table, we can also extrapolate another figure A.3 which shows that the execution time ascends exponentially with the increase of string length. In this figure, each line represents a different length of string $s_1$ which ranges from 50 bytes to 24000 bytes. $X$ axis represents the length of string $s_2$ which also ranges from 50 bytes to 24000 bytes and $Y$ axis represents the execution time used to compute the distance between two strings $s_1$ and $s_2$. We can see with the increase of length of string $s_1$ the execution time changes from linear growth to exponential growth. For example, when the length of string $s_1$ is below 10000 bytes, the execution time grows linearly with the increase of the length of string $s_2$, but when the length of string $s_1$ reaches 16000 bytes, the execution time grows exponentially with the increase of length of string $s_2$. And under our current computing environment, it takes average 1 minutes to compute three distance value between two executable files, that means it will take about 2 days to compare 100 samples.

In addition, this method require the computation of the distances between all pairs of functions, which invariably results in a computational complexity of $O(n^2)$. The reality is the rapidly increasing number of malware programs each day, so it is clear that one of the most important requirements for classification system is scalability, it should classify a large amount of malware in a reasonable time. In this method, I store all the information related to the function distance which is accompanied by large memory and processing overheads vital for the real time deployment.

### A.7 Summary

In this Appendix, I presented function distance based experiment. Hex format string was extracted from the binary code of a IDA function in our ida2DBMS, three distance algorithms were applied to these string information to acquire the similarity between any two functions, and then based on these similarity, similarity between two files was calculated. The experimental results and performance analysis showed that this function distance based method has computational complexity of $O(n^2)$. This method should be leveraged before being put into application.
Appendix B

Experimental Dataset

All of your test samples are from CA’s VET zoo (www.ca.com); thus they have been pre-classified using generally acceptable mechanical means.

Table 3.1 in Section 3.3.1 gives an overview of all the families in our experimental data set. There are 17 malware families and cleanware, the total number is 2939, including 12 Trojan families, 3 Virus families and 2 Worm families and 541 cleanware.

In this section, I give a detailed description of these families.

B.1 Adclicker

AdClicker is a detection name used by CA to identify of a family of malicious programs that are designed to artificially inflate the numbers of visitors to a given website by creating fake page views, to share the primary functionality of artificially generating traffic to pay-per-click Web advertising campaigns in order to create or boost revenue.

AdClickers typically copy themselves to a system folder in an attempt to remain inconspicuous, and create a load point so that they run every time Windows starts. The Trojans may also perform the following actions:

- Lower security settings.
- Attempt to download files, including other malware.
- Display messages and/or advertisements.
The Trojans typically then begin their routines to generate fake clicks. In general, Adclicker aim to execute without the knowledge of the user, but indicators of infection may include slow or jittery Internet browsing. In some cases the Trojans may consume significant bandwidth. Some variants of Trojans. Adclicker may additionally display messages and/or advertisements on the compromised computer. The most immediate risk to users is that the bandwidth being consumed by the threats, and user may be vulnerable to the effects of other malware that may be downloaded by the threats.

### B.2 Bancos

Bancos and Banker are two of the most prevalent banking Trojans that detected by CA since 2008. Bancos malicious programs run silently in the background to monitor web browser activities, sometime they imitate legitimate applications distributed by banks and there is no way a user can tell the difference between the real and fake graphical user interfaces. It can create fake login page for certain banking sites, after they get the control over the keyboard they can intercept login credentials entered on the website by the user, which is used for stealing user names and passwords which can be sent to the attacker via e-mail.

### B.3 Banker

Banker family is used to describe trojans which attempt to steal sensitive information that can be used to gain unauthorized access to bank accounts via Internet Banking.

### B.4 Gamepass

Gamepass is a family of Trojans that steals login credentials and in-game information related to various Massively Multiplayer Online Role Playing Games (MMORPG). Files belonging to this malware family are Win32 executables that are packed/protected using various packers such as UPX, UPack, FSG and NSAnti.
Gamepass Trojan variants steal sensitive information related to various MMORPGs and other online games, particularly those popular in China and East Asia. Some game titles that this Trojan family targets include:

- AskTao
- Nexia the Kingdom of the Wind
- MapleStory
- Dungeon & Fighter
- Fantasy Journey Online
- The Warlords
- World of Warcraft
- Perfect World
- Yulgang (Scions of Fate)
- Legend of Mir II
- Lineage II

Gamepass generally monitors window titles and processes, searching for indications that the targeted game has been launched. For instance, it is common for the Trojan to initialize its logging routines after it has found an active window with the title of the game, which is commonly in Chinese for most of the titles targeted.

It is also common for some Gamepass variants to drop a DLL which allows it to install either a keyboard or a mouse hook. The Trojan waits until the user has entered a keystroke or clicked a mouse button before it begins logging sensitive information. The Trojan logs the account name and password that the user enters into the game’s login prompt window in order to access their account.

Gamepass variants may also steal details specific to the host machine, as well as in-game information related to the game being played. In-game information is stolen by the Trojan in various ways, such as:

- By reading information from sub-windows accessed by the user in-game
- By reading the process memory of the game’s main executable
- By reading information from the game’s setup files.
Such information includes:

- IP and host name of machine
- Game server name
- Role information (character’s name, job/role, sex, level)
- Game information (amount of currency, map details)

Gamepass can store this information in a log file, and then send the log file to a remote attacker, either via email or by posting the information to a remote website.

### B.5 SillyDl

SillyDl variants may be installed via Internet Explorer exploits when users visit malicious web pages; other Trojan downloaders or components; or they may be packaged with software that the user has chosen to install. A downloader is a program that automatically downloads and runs and/or installs other software without the user’s knowledge or permission. In addition to downloading and installing other software, it may download updated versions of itself.

SillyDl variants may download other Trojans, or non-malicious programs such as adware. At any given moment in time, the program(s) it attempts to download may be changed or updated, or may be unavailable altogether. This family of Trojans usually downloads using HTTP.

### B.6 Vundo

Vundo is a large family of Trojans that contain backdoor functionality that gives an unauthorized user access to an affected machine. They have been associated with adware. Vundo variants typically use random filenames, however reports indicate that many variants of this increasingly large family are originally downloaded as bkinst.exe. Current Vundo variants reported from the wild copy themselves and drop an executable file in one or more of the following subdirectories of the %Windows% directory: “addins”, “AppPatch”, “assembly”, “Config”, “Cursors”, “Driver
“Registration”, “repair”, “security”, “ServicePackFiles”, “Speech”, “system”, “sys-
tem32”, “Tasks”, “Web”, “Windows Update Setup Files”, “Microsoft”.

Vundo variants modify the following registry entries to ensure that their created
copies execute at each Windows start:

- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
- HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
- HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Vundo also drops a DLL in the %Temp% directory using the filename of the
executable reversed with a .dat extension. For example, if the executable created is
wavedvd.exe, Vundo would drop dvdevaw.dat in the %Temp% directory.

This DLL is registered as a service process and is used to protect the main ex-
cutables. The DLL is injected randomly into any of the other running processes on
the system. Although the main process is visible, if the process is terminated, it will
be restarted from the system memory by the injected DLL process. Using the backup
copy of the executable stored in memory, the process is able to re-create any files
which are deleted.

The DLL also creates a BHO (Browser Helper Object) class in the registry that
may appear similar to the following (for example):

- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects\{68132581-
  10F2-416E-B188-4E648075325A}

The executable also create a configuration file in its current directory, using its
own filename backwards, with the extension .ini. For example: if the executable was
dropper.exe it would create a configuration file named reppord.ini. Vundo creates
backup copies of the configuration file (should it exist) using the same filename, but
with extensions .bak1 and .bak2.
B.7 Frethog

Frethog is a family of trojans that steals passwords for online games. When executed, Frethog copies itself to either the %Windows% or %Temp% directories using a variable filename that differs according to variant.

Frethog modifies the registry to ensure that this copy is executed at each Windows start, for example:

- HKLM\Software\Microsoft\Windows\CurrentVersion\Run\upxdnd = “%Temp%\upxdnd.exe”.

Recent Frethog variants often also drop a DLL file into the %Windows% or %Temp% directories.

B.8 SillyAutorun

SillyAutoRun is a worm that spreads via removable drives. The worm also targets Trend Micro’s OfficeScan product files and registry keys. SillyAutorun executes when a previously infected removable drive is enabled and “AutoPlay;; launches the worm. The worm checks the path %SysDrive%:Program Files\Trend Micro\OfficeScan\ in an attempt to detect whether Trend Micro’s OfficeScan product is installed. This would usually be C:\Program Files\Trend Micro\OfficeScan\.

If the product directory is found, the worm copies itself to %SysDrive%:Program Files\Trend Micro\OfficeScan\KOfcfpwSvcs.exe and creates the following registry key to ensure it loads on the next user logon:

- HKLM\Software\Microsoft\Windows\CurrentVersion\Run\KOfcfpwSvcs.exe = “%SysDrive%\Program Files\Trend Micro\OfficeScan\KOfcfpwSvcs.exe”.

If the product directory is not detected, the worm copies itself to %System%\KOfcfpwSvcs.exe and creates the following registry key to ensure it loads on the next user logon:

- HKLM\Software\Microsoft\Windows\CurrentVersion\Run\KOfcfpwSvcs.exe = “%System%\KOfcfpwSvcs.exe”.
B.9 Alureon

Alureon is a family of Trojans with a variety of components that can download and execute arbitrary files, hijack the browser to display fake web pages, and report affected user’s queries performed with popular search engines.

B.10 Bambo

Win32.Bambo is a family of trojans that contain advanced backdoor functionality that allows unauthorized access to and extended control of an affected machine. Variants of this trojan submitted to CA have been compressed with a number of different tools (including UPX, AS-Pack and FSG), and have varied in length between 15,360 and 53,248 bytes. Although the filenames change according to the specific variant, the trojans of the Bambo family usually install themselves in the following manner. When executed, the trojan either copies itself to the %System% directory using one of the following filenames:

- load32.exe
- vxdmgr32.exe
- dllreg.exe
- netda.exe
- netdc.exe

Bambo variants have been seen to perform a variety of actions. The most commonly seen payloads may include the following:

1. Socks Proxy: The trojan opens up a Socks proxy on port 2283, or on a random port between 2710 and 53710 (in later variants). This allows for the forwarding of Internet traffic. In the later variants certain websites are contacted, notifying them of the port on which the socks proxy is open.

2. FTP server: An FTP server is opened up by the trojan on ports 1000 or 10000, allowing for FTP access to the files on an affected machine.

3. Steals Sensitive Information: The trojan gathers information from the infected computer, such as: Clipboard data, Keylogs of sensitive information, IP address of the infected machine, Owner registration of the Windows product, Internet banking and Webmoney details, ICQ numbers, E-mail server names, port numbers and passwords from Protected Storage.

4. Backdoor Functionality: A backdoor is generally opened up on TCP port 1001, although in later variants the port may be randomly selected. This backdoor accepts commands for several functions, including: Execute local programs, Open the CD drive, Close the CD drive, Play a sound file, Display a message box, Capture an image of the user’s screen, Change the e-mail address that keystroke captures, etc are sent to.
5. Edits Hosts file: Some variants have been seen to edit the Windows hosts file.

Win32/Bambo are a family of trojans that being distributed via SMS text messages sent to mobile phones, enticing people to visit a malicious website. The messages may contain the following:

‘Thanks for subscribing to *****.com dating service. If you don’t unsubscribe you will be charged $2 per day’.

The text message then directs the recipient to visit a website in order to unsubscribe from the service and avoid being charged. This website contains a fake dating service page, which entices users to enter their phone number, at which point it attempts to load an executable file called ‘unregister.exe’. The web page instructs users to click the ‘Run’ button on each warning page that Windows displays, to allow the program to execute. If the program is run, it installs the Win32/Bambotrojan.

B.11 Boxed

Win32/Boxed is a varied family of trojans, some consisting of an IRC-based backdoor which can download and execute arbitrary files. Several of the components downloaded by this backdoor are also included in the Boxed family. Earlier variants of Boxed perform Denial of Service attacks against specific hosts. Most Boxed variants attempt to stop and/or delete the following services that belong to antivirus products, the Windows Update client and the Windows Firewall:

- wscsvc
- SharedAccess
- kavsve
- SAVScan
- Symantec Core LC
- navapsvc
- wuauserv

Some also delete the registry key and all subkeys and values at:

HKLM \ Software \ Microsoft \ Windows \ CurrentVersion \ Run \ KAVPersonalse50.
If the system is running Windows XP Service Pack 2 and the following registry entry is present:

\texttt{HKLM \ System \ CurrentControlSet \ Services \ SharedAccess \ Parameters \ FirewallPolicy \ StandardProfile \ AuthorizedApplications \ List.}

the trojan may attempt to give itself access through the Windows Firewall by creating a registry entry at:

\texttt{HKLM \ System \ CurrentControlSet \ Services \ SharedAccess \ Parameters \ FirewallPolicy \ StandardProfile \ AuthorizedApplications \ List \ (pathandfilenameoftrojan) = \langle pathoftrojan \\ \ (filenameoftrojan):*:enabled:Microsoft Update'.

\textbf{B.12 Clagger}

Win32/Clagger are a family of trojans that download files onto the affected system as well as terminating security related processes. The trojan has been distributed as an FSG-packed, Win32 executable that is between 5000 and 6000 bytes in length. The trojan’s primary function is to download and execute files (including additional malware) from a specific domain that is encrypted in the trojan’s code. Files are downloaded to the \texttt{%Windows}\footnote{\%Windows\% is a variable location. The malware determines the location of the current Windows folder by querying the operating system. The default installation location for the Windows directory for Windows 2000 and NT is C:\Winnt; for 95,98 and ME is C:\Windows; and for XP is C:/Windows.} directory using file names contained in the URL. After downloading, Clagger runs a batch file that deletes its executable. The trojan terminates the following processes if they are running on the affected system:

- firewall.exe
- MpfService.exe
- zonealarm.exe
- NPROTECT.exe
- kp4gui.exe
- atguard.exe
- tpfw.exe
- kp4ss.exe
- zapro.exe
The trojan adds its executable to the following registry entry so that it can be added as an exception to the Windows Firewall.

\[HKLM\ system\ currentControlset\ services\ sharedaccess\ parameters\ firewallpolicy\ standardprofile\ authorizedapplications\ list\]

**B.13 Robknot**

Robknot spreads via e-mail and modify system settings. When executed, Robknot copies itself a number of times to the ‘%Profiles% \ Local Settings \ Application Data’ directory. It copies itself using the following file names:

- CSRSS.EXE
- INETINFO.EXE
- LSASS.EXE
- SERVICES.EXE
- SMSS.EXE
- WINLOGON.EXE

It then executes all of these files, which in turn modify the registry so that they are executed at each Windows start:

\[HKCU\ Software\ Microsoft\ Windows\ CurrentVersion\ Run\ Tok-Cirrhatus = ‘%Profiles% \ Local Settings \ Application Data \ \langle filename.exe \rangle’\]

Robknot also copies itself to the ‘%Windows% \ ShellNew’ directory and sets the following registry value so that this copied file is executed at each Windows Start:

\[HKLM\ software\ microsoft\ windows\ currentversion\ run\ Bron-Spizaetus = ‘%Windows% \ ShellNew \ \langle filename.exe \rangle’\].

Robknot also searches for folders to copy itself to. If it finds an executable in a folder, it copies itself to the folder using the folder name. It replaces any executable
in the folder which has the same name as the folder it is in. Robknot variants use the folder icon which is trick. They also modify the following registry entry to hide all file extensions in the Explorer view; hence, Robknot variants appear to be a folder instead of an executable:

\[HKCU \makebox[0pt][l]{\backslash} \makebox[0pt][l]{software} \backslash \makebox[0pt][l]{microsoft} \backslash \makebox[0pt][l]{windows} \backslash \makebox[0pt][l]{currentversion} \backslash \makebox[0pt][l]{explorer} \backslash \makebox[0pt][l]{advanced} \backslash \text{HideFileExt} = 1\]

Robknot sends itself as an e-mail attachment to e-mail addresses harvested from the affected machine. It searches for e-mail addresses to send itself to in files that have the following extensions on the local system:

- TXT
- EML
- WAB
- ASP
- PHP
- CFM
- CSV
- DOC

Robknot modifies the desktop theme as well as modifying the following registry value to set Explorer to not show files with the 'Hidden' attribute:

\[HKCU \makebox[0pt][l]{\backslash} \makebox[0pt][l]{software} \backslash \makebox[0pt][l]{microsoft} \backslash \makebox[0pt][l]{windows} \backslash \makebox[0pt][l]{currentversion} \backslash \makebox[0pt][l]{explorer} \backslash \makebox[0pt][l]{advanced} \backslash \text{Hidden} = 0\]

Robknot modifies the following registry value to set Explorer to not show ""protected operating system files"" (i.e. - files with both the "‘System’" and "‘Hidden’ attributes set):

\[HKCU \makebox[0pt][l]{\backslash} \makebox[0pt][l]{software} \backslash \makebox[0pt][l]{microsoft} \backslash \makebox[0pt][l]{windows} \backslash \makebox[0pt][l]{currentversion} \backslash \makebox[0pt][l]{explorer} \backslash \makebox[0pt][l]{advanced} \backslash \text{ShowSuperHidden} = 0\]

Robknot also disables the Folder Options menu item in Explorer:

\[HKCU \makebox[0pt][l]{\backslash} \makebox[0pt][l]{software} \backslash \makebox[0pt][l]{microsoft} \backslash \makebox[0pt][l]{windows} \backslash \makebox[0pt][l]{currentversion} \backslash \makebox[0pt][l]{Policies} \backslash \makebox[0pt][l]{Explorer} \backslash \text{NoFolderOptions} = 1\]

Robknot attempts to disable the command prompt by modifying the following value:

\[HKCU \makebox[0pt][l]{\backslash} \makebox[0pt][l]{software} \backslash \makebox[0pt][l]{microsoft} \backslash \makebox[0pt][l]{windows} \backslash \makebox[0pt][l]{currentversion} \backslash \makebox[0pt][l]{Policies} \backslash \makebox[0pt][l]{System} \backslash \text{DisableCMD} = 0\]
Robknot disables registry tools such as Regedit.exe by modifying the following registry entry:

\[HKCU\ \backslash \ software \ \backslash \ microsoft \ \backslash \ windows \ \backslash \ currentversion \ \backslash \ Policies \ \backslash \ System \ \backslash \ DisableRegistryTools = 1\]

Robknot also modifies the file C:\autoexec.bat to include the line ‘pause’ so that the system pauses at each Windows start.

### B.14 Robzips

Robzips spreads via e-mail. They spread by sending a ZIP archive attached to an e-mail message. The ZIP archive contains a downloader and a batch file. When executed, Win32/Robzips creates a folder with a random name in the %System% directory and copies itself to this folder using the following file names:

- smss.exe
- csrss.exe
- lsass.exe
- services.exe
- winlogon.exe

It then executes all of these files. Robzips then makes the following registry modifications so that some of these copies are executed at each Windows start:

\[HKLM \ \backslash \ Software \ \backslash \ Microsoft \ \backslash \ Windows \ \backslash \ CurrentVersion \ \backslash \ Run \ \langle \ randomname \rangle = 'C:\\langle \ randomfoldername \rangle \langle \ randomname \rangle .exe'\]

\[HKCU \ \backslash \ Software \ \backslash \ Microsoft \ \backslash \ Windows \ \backslash \ CurrentVersion \ \backslash \ Run \ \langle \ randomname \rangle = '\%System\%\\langle \ randomfoldername \rangle \langle \ randomname \rangle .exe'\]

Some variants also make the following registry modifications:

\[HKCU \ \backslash \ Software \ \backslash \ Microsoft \ \backslash \ Windows \ \backslash \ CurrentVersion \ \backslash \ Run \ \langle \ randomname \rangle = 'C:\Baca \ Bro \ !!!.txt'\]

\[HKLM \ \backslash \ Software \ \backslash \ Microsoft \ \backslash \ Windows \ \backslash \ CurrentVersion \ \backslash \ Run \ \langle \ randomname \rangle = 'C:\Windows\langle \ randomname \rangle .exe'\]

Most Robzips variants create a text file ‘C:\Baca Bro !!!!.txt’ containing the following text: When the user opens this file, Robzips closes it and then displays the following messages (see Figure robzips-textcontaining on a console window: Robzips sends e-mail to e-mail addresses harvested from files located on the local hard drive. It searches through files that have the following extensions:

\[2\%System\% is variable location. The malware determines the location of these folders by querying the operating system. The default installation location for the System directory for Windows 2000 and NT is C:\Winnt\System32; for 95,98 and ME is C:\Windows\System; and for XP is C:\Windows\System32. The default installation location for the Windows directory for Windows 2000 and NT is C:\Winnt; for 95,98 and ME is C:\Windows; and for XP is C:\Windows.\]
Figure B.1. Text File Created by Robzips

Figure B.2. Messages Displayed on a Console Window by Robzips
Figure B.3. Messages Displayed on a Console Window by Robzips

- asp
- cfm
- csv
- doc
- eml
- htm
- html
- php
- ppt
- txt
- wab
- xls

Robzips stops a number of applications from running at each Windows start by deleting their registry values (listed below) from the keys:

HKCU \ Software \ Microsoft \ Windows \ CurrentVersion \ Run

HKLM \ Software \ Microsoft \ Windows \ CurrentVersion \ Run

239
Robzips modifies the desktop theme, as well as modifying the following registry value so that Explorer does not show files with the ‘Hidden’ attribute:

```
HKCU \ Software \ Microsoft \ Windows \ CurrentVersion \ Explorer \ Advanced \ Hidden = 0
```

Robzips modifies the following registry value so that Explorer does not show ”‘Protected operating system files’” - that is, files with both ”‘System’” and ”‘Hidden’” attributes set:

```
HKCU \ Software \ Microsoft \ Windows \ CurrentVersion \ Explorer \ Advanced \ ShowSuperHidden = 0
```

## B.15 Looked

Looked is a family of file-infecting worms that spread via network shares. They also drop a DLL which is used to periodically download and execute arbitrary files. A minority of variants do not drop the DLL and behave slightly differently to the rest of this family. Recent examples of these different variants (at the time of publication) include Win32/Looked.EK and Win32/Looked.FU.

When executed, Win32/Looked copies itself to %Windows%\Logo1.exe and %Windows%\uninstall\rundl132.exe. It then executes the copy at %Windows%\Logo1.exe. When this is complete the worm uses a batch script to either delete the original file, or, if the original file was an infected executable, to replace this file with a clean copy, and then execute the clean file. Some variants copy themselves to %Windows%\rundl132.exe instead of %Windows%\uninstall\rundl132.exe.

Looked generally creates the following registry entry to ensure that the worm runs on system startup:

- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\load = “%Windows%\uninstall\rundl132.exe”

One of the following registry entries may be used by some variants instead of the entry above:

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\ load = “%Windows%\rundl132.exe”.
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\load = “%Windows%\rundl132.exe”

It drops a DLL to one of the following locations:

- %Windows%\RichDll.dll.
and injects code into the explorer.exe or Iexplore.exe processes so that the DLL’s code is executed. This code is used to download and execute arbitrary files, as described in the Payload section below. This action may result in an extra Windows Explorer window being opened shortly after the infected file is first run. Some variants also drop a second DLL with a random filename to the %Temp% directory. Many variants are packed using the NSAnti packer. For some of these variants, a driver file is also dropped to %Temp%\1 - 4 random alphanumeric characters\sys, which is later moved to %System%\wincab.sys. This file appears to be a component of the unpacker, and is used to hide some of the worm’s activities on the affected system.

B.16 Emerleox

Emerleox is a family of worms that spread via network shares and file infection. They can also download and execute arbitrary files and terminate processes and services on the affected system. Some Emerleox variants spread by infecting Win32 executables. Initially, Emerleox parses the affected machine looking for executable and/or HTML files. When an infected file is executed, the worm writes the original (uninfected file) to a file with the same filename but with an additional .exe extension; creates a batch file that deletes the infected file that was executed; renames the original to remove the additional .exe extension. If successful, the worm drops a copy of itself to a shared directory using a variable filename (for example: GameSetup.exe). It then tries to add a scheduled job to run this copy on the newly compromised system.

When executed Emerleox usually copies itself to the %System% or %System%\drivers directories. It often uses the filenames: “%System%\spoclsv.exe” or “%System%\drivers\svohost.exe”.

The worm then modifies the registry so that the main executable runs at each Windows start, for example:

- HKCU\Software\Microsoft\Windows\CurrentVersion\Run\svcshare = “%System%\drivers\spoclsv.exe”.
- HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\SoundMam = “%System%\svohost.exe”.

241
Agobot is an IRC controlled backdoor that can be used to gain unauthorized access to a victim’s machine. It can also exhibit worm-like functionality by exploiting weak passwords on administrative shares and by exploiting many different software vulnerabilities, as well as backdoors created by other malware. There are hundreds of variants of Agobot, and others are constantly being developed. The source code has been widely distributed, which has led to different groups creating modifications of their own. However, their core functionality is quite consistent. When first run, an Agobot will usually copy itself to the System directory. The file name is variable. It will also add registry entries to run this copy at Windows start, usually to these keys:

- HKLM\Software\Microsoft\Windows\CurrentVersion\Run.
- HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices.

For example, one variant observed “in the wild”, copies itself to: %System%\aim.exe and adds these values to the registry:

- HKLM\Software\Microsoft\Windows\CurrentVersion\Run\AOL Instant Messenger = “aim.exe”.
- HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices\AOL Instant Messenger = “aim.exe”.

242
Appendix C

Function Length Extraction Procedure

This section presents the opcode of stored procedures and functions used to extract function length features from Ida2DB. As we see from Figure C.1 that the procedure composed of four blocks and involved five tables in our Ida2Db, they are Instructions, Basic_blocks, functions, modules and Fun.Len_Table. In Appendix D we give the detailed information of our Ida2DB schema.

In our ida2DB schema, each entry in the functions table is composed of many entries in the basic_blocks table, and each entry in the basic_blocks table is composed of many entries in the instruction table, so our function length information extraction process is based on this structure. Database stored procedure "Fun.Len.Family" fetches length information of all the functions of all the modules from the specific family by invoking the Database stored procedure "Fun.Len.Module" which fetches length information of all the functions of the specific module. Database stored procedure "Fun.Len.Module" fetches each function length information by invoking the Database Internal Function "GetFun". Just as we talked before, each function is composed of many basic blocks, so in "GetFun", database Internal Function "GetBasicData" is invoked circularly to get all the data information of all the basic blocks of the specific function. Similarly in "GetBasicData" each instruction is traversed to fetch all the instruction data information of the assigned basic block.
C.1 Database Stored Procedure: Fun_Len_Family

Database stored procedure "Fun_Len_Family" fetches

```
USE [ida2DB]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[Fun_Len_Family]
@family nvarchar(100)
AS
BEGIN
SET NOCOUNT ON;
declare @mid bigint
declare cc cursor for select module_id from [dbo].[modules]
where family = @family
order by module_id
open cc
fetch next from cc into @mid
while (@@fetch_status=0)
```
C.2 Database Stored Procedure: Fun_Len_Module

USE [ida2DB]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE [dbo].[Fun_Len_Module]
@module_id bigint
AS
BEGIN
SET NOCOUNT ON;
declare @mid bigint, @function_id bigint, @fun_len bigint
declare @stemp nvarchar(max)
declare kk cursor for select module_id, function_id from 
[dbo].[functions]
where module_id = @module_id
order by function_id
open kk
fetch next from kk into @mid,@function_id
while (@@fetch_status=0)
begin
    set @stemp =''
    set @stemp = [dbo].[GetFun](@mid,@function_id)
    set @fun_len = len(@stemp)
If @fun_len >0
INSERT INTO fun_len (module_id, function_id, fun_len)
VALUES (@mid, @function_id, @fun_len);
    fetch next from kk into @mid, @function_id
end
close kk
deallocate kk
END

C.3 Database Internal Function: GetFun

USE [ida2DB]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE function [dbo].[GetFun]
(@module_id bigint = 0, @function_id bigint = 0)
RETURNS nvarchar(max)
AS
BEGIN
declare @s nvarchar(max)
set @s = ''
declare @pf bigint
select @pf = address from [dbo].[functions]
where module_id = @module_id and function_id = @function_id
declare @stemp nvarchar(max)
declare @mid bigint, @basic_id bigint
declare kk cursor for select module_id, id from [dbo].[basic_blocks]
where module_id = @module_id and parent_function = @pf
open kk
fetch next from kk into @mid,@basic_id
while (@@fetch_status=0)
  begin
    set @stemp ='
    set @stemp = [dbo].[GetBasicData](@module_id,@basic_id)
    set @s =@s + @stemp
    fetch next from kk into @mid,@basic_id
  end
close kk
deallocate kk
RETURN  @s
END

C.4 Database Internal Function: GetBasicData

USE [ida2DB]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
Create FUNCTION [dbo].[GetBasicData]
(@module_id bigint, @basic_block_id bigint)
RETURNS nvarchar(max)
AS
BEGIN
  DECLARE @Result nvarchar(max)
  set @Result=''
  SELECT @Result = @Result+data
  FROM [dbo].[instructions]
  where module_id =@module_id and basic_block_id =@basic_block_id
RETURN @Result
END
Appendix D

Ida2DB schema

D.1 Basic Tables

<table>
<thead>
<tr>
<th>No.</th>
<th>Table_Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Modules</td>
<td>Basic information of analysed object</td>
</tr>
<tr>
<td>2</td>
<td>Functions</td>
<td>All the functions, including the import functions</td>
</tr>
<tr>
<td>3</td>
<td>Basic_blocks</td>
<td>Basic blocks in functions</td>
</tr>
<tr>
<td>4</td>
<td>Instructions</td>
<td>Instructions at some address</td>
</tr>
<tr>
<td>5</td>
<td>Callgraph</td>
<td>Callers and Callees</td>
</tr>
<tr>
<td>6</td>
<td>Control_flow_graph</td>
<td>Links between basic blocks</td>
</tr>
<tr>
<td>7</td>
<td>Operand_strings</td>
<td>The operand strings as shown by IDA</td>
</tr>
<tr>
<td>8</td>
<td>Expression_tree</td>
<td>Expressions composing the operands as a tree</td>
</tr>
<tr>
<td>9</td>
<td>Operand_tuples</td>
<td>Maps addresses to the operands used by the instruction at such location</td>
</tr>
<tr>
<td>10</td>
<td>Expression_substitutions</td>
<td>Allows to replace any part of the expression tree of an operand with a string, variable names are handled through this table</td>
</tr>
<tr>
<td>11</td>
<td>Operand_expressions</td>
<td>Relates the operands to the expressions composing them</td>
</tr>
<tr>
<td>12</td>
<td>Address_references</td>
<td>Contains all references, both to code and data labeled with their type</td>
</tr>
<tr>
<td>13</td>
<td>Address_comments</td>
<td>Comment String Information</td>
</tr>
<tr>
<td>14</td>
<td>Sections</td>
<td>Sections information</td>
</tr>
<tr>
<td>15</td>
<td>Strings_windows</td>
<td>String window of IDA</td>
</tr>
<tr>
<td>16</td>
<td>Function_length</td>
<td>Function length information</td>
</tr>
<tr>
<td>17</td>
<td>Statistic_table</td>
<td>Statistical information</td>
</tr>
<tr>
<td>18</td>
<td>FileDateTime</td>
<td>Date and time information of executable files</td>
</tr>
</tbody>
</table>

Table D.1. Basic Tables
### D.2 Tables Description

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module_id</td>
<td>int</td>
<td>identity</td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Name</td>
<td>nvarchar</td>
<td>256</td>
<td>No</td>
<td>Real name of the analysed file</td>
</tr>
<tr>
<td>Md5</td>
<td>nchar</td>
<td>32</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Sha1</td>
<td>nchar</td>
<td>40</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>nvarchar</td>
<td>1000</td>
<td>Yes</td>
<td>Comment</td>
</tr>
<tr>
<td>Entry_point</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The entry point of the executable</td>
</tr>
<tr>
<td>Import_time</td>
<td>datetime</td>
<td></td>
<td>No</td>
<td>The time when the file being analysed</td>
</tr>
<tr>
<td>Filetype</td>
<td>nvarchar</td>
<td>256</td>
<td>No</td>
<td>The type of file</td>
</tr>
<tr>
<td>Platform</td>
<td>nvarchar</td>
<td>256</td>
<td>No</td>
<td>The platform of file</td>
</tr>
<tr>
<td>Family</td>
<td>nvarchar</td>
<td>256</td>
<td>No</td>
<td>The family information</td>
</tr>
<tr>
<td>Variant</td>
<td>nvarchar</td>
<td>256</td>
<td>No</td>
<td>The variant information of the file</td>
</tr>
<tr>
<td>PK</td>
<td>Module_id</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table D.2. Modules
<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function_id</td>
<td>bigint</td>
<td>Identity</td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Section_name</td>
<td>nvarchar</td>
<td>256</td>
<td>No</td>
<td>The section name that function belongs to</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The start address of function</td>
</tr>
<tr>
<td>Name</td>
<td>nvarchar</td>
<td>1000</td>
<td>No</td>
<td>The name of function</td>
</tr>
<tr>
<td>Function_type</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The type of function</td>
</tr>
<tr>
<td>Name_md5</td>
<td>nchar</td>
<td>32</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Cyclomatic_complexity</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>Complexity of function</td>
</tr>
</tbody>
</table>

**PK**
- Module_id + address

**FK**
- Module_id(references modules:module_id)
  - function_standard=0
  - function_library=1
  - function_imported=2
  - function_thunk=3
  - function_exported=4

Table D.3. Functions

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic_block_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Id</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The no of basic_block</td>
</tr>
<tr>
<td>Parent_function</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The function address that this basic block belongs to</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The start address of this basic block</td>
</tr>
</tbody>
</table>

**PK**
- Basic_block_id

**Unique**
- Module_id + id + parent_function

**FK**
- Module_id(references modules:module_id)

Table D.4. Basic_blocks
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Basic_block_id</td>
<td>bigint</td>
<td></td>
<td></td>
<td>The no of basic block that this instruction belongs to</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The address of this instruction</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>nvarchar</td>
<td>32</td>
<td>Yes</td>
<td>Mnemonic</td>
</tr>
<tr>
<td>Sequence</td>
<td>int</td>
<td></td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>Data</td>
<td>nvarchar(MAX)</td>
<td></td>
<td>Yes</td>
<td>Reserved</td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td>Instruction_id</td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
<td>Module_id + basic_block_id + address</td>
</tr>
<tr>
<td>FK</td>
<td></td>
<td></td>
<td></td>
<td>Module_id(references modules:module_id)</td>
</tr>
</tbody>
</table>

Table D.5. Instructions

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callgraph_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>The no of callgraph</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Src</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The start address of caller</td>
</tr>
<tr>
<td>Src_basic_block_id</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>No of basic block of caller that revokes</td>
</tr>
<tr>
<td>Src_address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The address of instruction that revokes</td>
</tr>
<tr>
<td>Dst</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The start address of callee</td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td>Callgraph_id</td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
<td>Module_id + Callgraph_id</td>
</tr>
<tr>
<td>FK</td>
<td></td>
<td></td>
<td></td>
<td>Module_id(references modules:module_id)</td>
</tr>
</tbody>
</table>

Table D.6. Callgraph
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control_flow_graph_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>The no of Control_flow_graph</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Parent_function</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The start address of function that this control flow graph belongs to</td>
</tr>
<tr>
<td>Src</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The no of source basic block</td>
</tr>
<tr>
<td>Dst</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Kind</td>
<td>int</td>
<td></td>
<td>NO</td>
<td>Kind</td>
</tr>
</tbody>
</table>

PK  
Unique  
FK  
Kind values:

<table>
<thead>
<tr>
<th>Kind values:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Control_flow_graph_id</td>
</tr>
<tr>
<td></td>
<td>Module_id + Control_flow_graph_id</td>
</tr>
<tr>
<td></td>
<td>Module_id(references modules:module_id)</td>
</tr>
<tr>
<td></td>
<td>branch_type_true=0</td>
</tr>
<tr>
<td></td>
<td>branch_type_false=1</td>
</tr>
<tr>
<td></td>
<td>branch_type_unconditional=2</td>
</tr>
<tr>
<td></td>
<td>branch_type_switch=3</td>
</tr>
</tbody>
</table>

Table D.7. Control_flow_graph

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand_string_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Str</td>
<td>nvarchar</td>
<td>1000</td>
<td>Yes</td>
<td>Expression of operand</td>
</tr>
</tbody>
</table>

PK  
Unique  
FK  

<table>
<thead>
<tr>
<th>PK</th>
<th>Operand_string_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique</td>
<td>Module_id + Operand_string_id</td>
</tr>
<tr>
<td>FK</td>
<td>Module_id(references modules:module_id)</td>
</tr>
</tbody>
</table>

Table D.8. Operand_strings
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression_tree_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Expr_type</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Type of expression</td>
</tr>
<tr>
<td>Symbol</td>
<td>nvarchar</td>
<td>256</td>
<td>Yes</td>
<td>Symbol of expression</td>
</tr>
<tr>
<td>Immediate</td>
<td>bigint</td>
<td></td>
<td>Yes</td>
<td>Immediate</td>
</tr>
<tr>
<td>position</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Reserved</td>
</tr>
<tr>
<td>parent_id</td>
<td>bigint</td>
<td></td>
<td>Yes</td>
<td>The no of parent node</td>
</tr>
</tbody>
</table>

**PK**

| Unique                | Module_id + Expression_tree_id |
| FK                    | Module_id(references modules:module_id) |

**Expr_type values:**

- node_type_mnemonic_id=0
- node_type_symbol_id=1
- node_type_immediate_int_id=2
- node_type_immediate_float_id=3
- node_type_operator_id=4

Table D.9. Expression_tree

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand_tuple_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The address of instruction that operand belongs to</td>
</tr>
<tr>
<td>Operand_id</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The no of operand</td>
</tr>
<tr>
<td>Position</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The position of operand in an instruction,starts from 0</td>
</tr>
</tbody>
</table>

**PK**

| Unique                | Operand_tuple_id |
| FK                    | Module_id(references modules:module_id) |

Table D.10. Operand_tuples

253
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression_substitution_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The address of instruction</td>
</tr>
<tr>
<td>Operand_id</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The no of operand</td>
</tr>
<tr>
<td>Expr_id</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The no of expression of that operand</td>
</tr>
<tr>
<td>Replacement</td>
<td>nvarchar</td>
<td>1000</td>
<td>Yes</td>
<td>The new one</td>
</tr>
</tbody>
</table>

PK
Expression_substitution_id

Unique
Module_id + Expression_substitution_id

FK
Module_id(references modules:module_id)

Table D.11. Expression_substitution

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand_expression_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Operand_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The no of operand</td>
</tr>
<tr>
<td>Expr_id</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The no of expressions that compose the operand</td>
</tr>
</tbody>
</table>

PK
Operand_expression_id

Unique
Module_id + Operand_expression_id

FK
Module_id(references modules:module_id)

Table D.12. Operand_expression

254
<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_reference_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The address of source instruction</td>
</tr>
<tr>
<td>Target</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>The address of destination instruction</td>
</tr>
<tr>
<td>Kind</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The no of expression</td>
</tr>
</tbody>
</table>

PK

Unique

FK

Kind values:
- conditional_branch_true=0
- conditional_branch_false=1
- unconditional_branch=2
- branch_switch=3
- call_direct=4
- call_indirect=5
- call_indirect_virtual=6
- data=7
- data_string=8

Table D.13. Address_reference

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address_comments_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>Address information</td>
</tr>
<tr>
<td>Comment</td>
<td>nvarchar</td>
<td>1000</td>
<td>Yes</td>
<td>Comment</td>
</tr>
</tbody>
</table>

PK

FK

Address_comments_id

Module_id(references modules:module_id)

Table D.14. Address_comments
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Name</td>
<td>nchar</td>
<td>256</td>
<td>No</td>
<td>Section name</td>
</tr>
<tr>
<td>Base</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>Base address of section</td>
</tr>
<tr>
<td>Start_address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>Start address of section</td>
</tr>
<tr>
<td>End_address</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>End address of section</td>
</tr>
<tr>
<td>Length</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Length of section data</td>
</tr>
<tr>
<td>Data</td>
<td>nvarchar</td>
<td>yes</td>
<td></td>
<td>Data(original format)</td>
</tr>
</tbody>
</table>

| PK          | Section_id |
| Unique      | Module_id + name |
| FK          | Module_id(references modules:module_id) |

Table D.15. Sections

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>String_window_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Section_name</td>
<td>nvarchar</td>
<td>30</td>
<td>yes</td>
<td>The section name that function belongs to</td>
</tr>
<tr>
<td>Address</td>
<td>bigint</td>
<td></td>
<td>yes</td>
<td>address of string</td>
</tr>
<tr>
<td>Strlength</td>
<td>bigint</td>
<td></td>
<td>yes</td>
<td>length of string</td>
</tr>
<tr>
<td>Strtype</td>
<td>Int</td>
<td></td>
<td>yes</td>
<td>type of string</td>
</tr>
<tr>
<td>String data</td>
<td>nvarchar</td>
<td>MAX</td>
<td>Yes</td>
<td>String information</td>
</tr>
</tbody>
</table>

| PK             | String_window_id |
| Unique         | Module_id + Section_name + address |
| FK             | Module_id(references modules:module_id) |

Strtype values:
- C=0
- Pascal=1
- Pascal,2 byte length=2
- Unicode=3
- Pascal,4 byte length=4
- Pascal style Unicode,2 byte length=5
- Pascal style Unicode,4 byte length=6

Table D.16. Strings_window
<table>
<thead>
<tr>
<th><strong>Column name</strong></th>
<th><strong>Type</strong></th>
<th><strong>Length</strong></th>
<th><strong>Is Null</strong></th>
<th><strong>Explanation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function_id</td>
<td>bigint</td>
<td>identity</td>
<td>No</td>
<td>Record no</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Funlen</td>
<td>bigint</td>
<td></td>
<td>No</td>
<td>Function length in bytes</td>
</tr>
</tbody>
</table>

**Unique**

Module_id + Function_id

**FK**

Module_id(references modules:module_id)

Table D.17. Function_length

<table>
<thead>
<tr>
<th><strong>Column name</strong></th>
<th><strong>Type</strong></th>
<th><strong>Length</strong></th>
<th><strong>Is Null</strong></th>
<th><strong>Explanation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistic_table_id</td>
<td>int</td>
<td>identity</td>
<td>No</td>
<td>Record no</td>
</tr>
<tr>
<td>Module_id</td>
<td>int</td>
<td></td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>Num_function</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The number of functions the analysed file</td>
</tr>
<tr>
<td>Num_import_function</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The number of import functions of the analysed file</td>
</tr>
<tr>
<td>Num_basic_block</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The number of basic blocks of the analysed file</td>
</tr>
<tr>
<td>Sum_cc</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The sum value of Cyclomatic complexity of all the functions of the analysed file</td>
</tr>
<tr>
<td>Max_cc</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The maximum value of Cyclomatic complexity of all the functions of the analysed file</td>
</tr>
<tr>
<td>Min_cc</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The minimum value of Cyclomatic complexity of all the functions of the analysed file</td>
</tr>
<tr>
<td>Avg_cc</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The average value of Cyclomatic complexity of all the functions of the analysed file</td>
</tr>
<tr>
<td>Num_instruction</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The number of instructions of the analysed file</td>
</tr>
<tr>
<td>OEP</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The original entry point of the analysed file</td>
</tr>
<tr>
<td>Num_section</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The number of sections of the analysed file</td>
</tr>
<tr>
<td>Num_string</td>
<td>int</td>
<td></td>
<td>No</td>
<td>The number of strings of the analysed file</td>
</tr>
</tbody>
</table>

**PK**

Statistic_table_id

**FK**

Module_id(references modules:module_id)

Table D.18. Statistic_table
<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Length</th>
<th>Is Null</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module_id</td>
<td>int</td>
<td>identity</td>
<td>No</td>
<td>Record no of the analysed file</td>
</tr>
<tr>
<td>name</td>
<td>nvarchar</td>
<td>256</td>
<td>no</td>
<td>Real name of the analysed file</td>
</tr>
<tr>
<td>md5</td>
<td>nchar</td>
<td>32</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Sha1</td>
<td>nchar</td>
<td>40</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>nvarchar</td>
<td>1000</td>
<td>Yes</td>
<td>Comment</td>
</tr>
<tr>
<td>entry_point</td>
<td>bigint</td>
<td></td>
<td>no</td>
<td>The entry point of the executable</td>
</tr>
<tr>
<td>Import_time</td>
<td>datetime</td>
<td></td>
<td>no</td>
<td>The time when the file being analysed</td>
</tr>
<tr>
<td>Filetype</td>
<td>nvarchar</td>
<td>256</td>
<td>no</td>
<td>The type of file</td>
</tr>
<tr>
<td>Platform</td>
<td>nvarchar</td>
<td>256</td>
<td>no</td>
<td>The platform of file</td>
</tr>
<tr>
<td>Family</td>
<td>nvarchar</td>
<td>256</td>
<td>no</td>
<td>The family information</td>
</tr>
<tr>
<td>Variant</td>
<td>nvarchar</td>
<td>256</td>
<td>no</td>
<td>The variant information of the file</td>
</tr>
<tr>
<td>FTimeStamp</td>
<td>nvarchar</td>
<td>256</td>
<td>no</td>
<td>The date time information of the file</td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td>Module_id</td>
</tr>
</tbody>
</table>

Table D.19. Filedatetime