# Counter Intrusion Software

## Malware Detection using Structural and Behavioural Features and Machine Learning

by

Joseph Rabaiotti
School of Computer Science
Cardiff University

A thesis submitted in partial fulfilment of the requirement for the degree of Doctor of Philosophy

Version: August 4, 2007

UMI Number: U584957

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U584957
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

**DECLARATION**

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed ………………………………………………… (candidate)    Date 06/09/2007

**STATEMENT 1**

This thesis is being submitted in partial fulfillment of the requirements for the degree of ………………………… (~~insert MCh, MD, MPhil~~, PhD ~~etc, as appropriate~~)

Signed ………………………………………………… (candidate)    Date 06/09/2007

**STATEMENT 2**

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references.

Signed ………………………………………………… (candidate)    Date 06/09/2007

**STATEMENT 3**

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ………………………………………………… (candidate)    Date 06/09/2007

# Abstract

Over the past twenty-five years malicious software has evolved from a minor annoyance to a major security threat. Authors of malicious software are now more likely to be organised criminals than bored teenagers, and modern malicious software is more likely to be aimed at stealing data (and hence money) than trashing data. The arms race between malware authors and manufacturers of anti-malware software continues apace, but despite this, the majority of anti-malware solutions still rely on relatively old technology such as signature scanning, which works well enough in the majority of cases but which has long been known to be ineffective if signatures are not updated regularly.

The need for regular updating means there is often a critical window between the publication of a flaw exploitable by malware and the distribution of the appropriate counter measures or signature. At this point a user system is open to attack by hitherto unseen malware. The object of this thesis is to determine if it is practical to use machine learning techniques to abstract generic structural or behavioural features of malware which can then be used to recognise hitherto unseen examples.

Although a sizeable amount of research has been done on various ways in which malware detection might be automated, most of the proposed methods are burdened by excessive complexity. This thesis looks specifically at the possibility of using learning systems to classify software as malicious or nonmalicious based on *easily-collectable* structural or behavioural data. On the basis of the experimental results presented herein it may be concluded that classification based on such structural data is certainly possible, and on behavioural data is at least feasible.

# Acknowledgements

### Personal acknowledgements

I would like to thank my supervisor Professor Antonia Jones for her support and encouragement during the past three years. I must also thank my friend Stuart Goring, without whose keen observations Chapter 3 would not have come into being, my parents Robert and Carole for supporting me financially and providing me with accommodation for the first two and a half years, and Mark, for supporting and putting up with me for the remainder of the time and without whose love and encouragement this thesis could never have been written.

### Technical acknowledgements

I must also acknowledge the significant contribution to this work made by the authors of the website which hosted the virus/malware library, without which this work could not have proceeded. I hope they will forgive me for not naming them explicitly, but they probably know who they are.

Work of this type is not without risk and approaches a grey area of UK Law. Everyone involved has taken great pains to remain within the law as we understand it. Considerable thought and discussion occurred within the School and University, before agreements were reached which ensured that the work could be undertaken safely, i.e. that no harm would be caused by the study. A number of precautions were put in place in order to isolate, and limit access to, the computers used in this work. Therefore, thanks are also due to my Supervisor Professor Antonia J. Jones, the Head of the Cardiff School of Computer Science Professor Nick Fiddian, Robert Evans of the School's Computer Support Group, to the University level Computer Support team in INSRV, and to the Corporate Compliance and HR sections in the University,[1] who collectively, and somewhat courageously, trusted me sufficiently to allow the work to go forward.

[1] http://www.cf.ac.uk/az/divisions/index.html

# Contents

Contents 3

List of Figures 9

List of Tables 11

1 Introduction 14

1.1 What Is Malicious Software? . . . . . . . . . . . . . . . . . . . . . . . 14
1.2 The Connectivity Issue . . . . . . . . . . . . . . . . . . . . . . . . . 15
1.3 Combat Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . 19
1.4 Scope and Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
1.4.1 Hypotheses . . . . . . . . . . . . . . . . . . . . . . . . . . 20
1.4.2 Aims . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
1.4.3 Principal Contributions . . . . . . . . . . . . . . . . . . . . 21

2 Historical Background and Literature Review 22

2.1 Types of Malware . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.1.1 Viruses . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.1.2 Worms . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.1.3 Other Email Tricks . . . . . . . . . . . . . . . . . . . . . . 28
2.1.4 Trojan Horses . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.1.5 Backdoors . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.1.6 Spyware and Adware . . . . . . . . . . . . . . . . . . . . . 30
2.1.7 Botnets and DDoS Attacks . . . . . . . . . . . . . . . . . . 32
2.1.8 Rootkits . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.1.9 Malware Payloads . . . . . . . . . . . . . . . . . . . . . . . 35

2.2 Vulnerabilities exploited by malware . . . . . . . . . . . . . . . . . . . . 36
2.2.1 Code Injection . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.2.2 Buffer Overflow . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.3 Malware Naming Conventions . . . . . . . . . . . . . . . . . . . . . . . 37
2.4 Detection and Prevention Techniques . . . . . . . . . . . . . . . . . . . 38
2.4.1 Heuristic Scanners . . . . . . . . . . . . . . . . . . . . . . . . 40
2.4.2 Structure-Based Detection . . . . . . . . . . . . . . . . . . . . 41
2.4.3 Activity Monitors . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.4.4 Behaviour-Based Anomaly Detection . . . . . . . . . . . . . . 42
2.5 Detection Countermeasures . . . . . . . . . . . . . . . . . . . . . . . . 42
2.5.1 Heuristic Frustration: Concealment . . . . . . . . . . . . . . . 43
2.5.2 Polymorphism and Metamorphism . . . . . . . . . . . . . . . 43
2.5.3 Encryption and Packing . . . . . . . . . . . . . . . . . . . . . 43
2.5.4 Retroviruses . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
2.6 Malware and the Law . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
2.7 Literature Review . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
2.7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
2.7.2 Early Research and Theory . . . . . . . . . . . . . . . . . . . 46
2.7.3 Practical Detection Systems . . . . . . . . . . . . . . . . . . . 48
2.7.4 Static and Semantic Analysis . . . . . . . . . . . . . . . . . . 49
2.8 Tools and Sources of Information . . . . . . . . . . . . . . . . . . . . 54

**3 The law of unintended consequences: a case study 55**

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.2 Disclaimer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.3 Unintended consequences . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.4 The login procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.5 The Vulnerability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.6 Attack Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
3.7 Probabilistic Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.8 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

**4 Machine learning classification methods and systems** **66**

4.1 Introduction . . . . . 66

4.2 Classification methods . . . . . 66

4.2.1 Decision Trees . . . . . 67

4.2.2 Naive Bayes . . . . . 68

4.2.3 Artificial Neural Networks . . . . . 69

4.3 Classifier Implementations: Machine Learning Systems . . . . . 70

4.3.1 C4.5 . . . . . 71

4.3.2 WEKA . . . . . 73

4.4 Conclusion . . . . . 74

**5 Gathering structural and behavioural data in Windows** **76**

5.1 Introduction . . . . . 76

5.2 Structural Data . . . . . 77

5.2.1 What sort of data to use? . . . . . 77

5.2.2 The Portable Executable (PE) file format . . . . . 77

5.2.3 The PE Header . . . . . 79

5.2.4 Sections and the Section Table . . . . . 79

5.2.5 Viruses and the PE format . . . . . 80

5.2.6 Collecting Structural Data . . . . . 87

5.3 Behavioural Monitoring . . . . . 88

5.3.1 The collection of real-time process data . . . . . 88

5.3.2 Behavioural Data Gathering Methods . . . . . 97

5.3.3 Two Means of Monitoring . . . . . 99

5.3.4 Windows Performance Data . . . . . 100

5.3.5 Lower-Level Methods . . . . . 102

5.3.6 Collecting Other Relevant Behavioural Data . . . . . 103

5.3.7 Behavioural Monitoring: Conclusion . . . . . 104

**6 Structural identification experiments** **106**

6.1 Introduction . . . . . 106

6.2 Methodology . . . . . 106

6.2.1 The data-gathering program . . . . . 107

6.2.2 The source datasets . . . . . 109

6.2.3 Creating experimental datasets . . . . . . . . . . . . . . . . . . . 110
6.2.4 Splitting into Training and Test sets . . . . . . . . . . . . . . . . 110
6.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
6.3.1 The kappa statistic $\kappa$ . . . . . . . . . . . . . . . . . . . . . . 111
6.3.2 Baseline . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
6.3.3 Learning curves . . . . . . . . . . . . . . . . . . . . . . . . 111
6.3.4 Experiment One . . . . . . . . . . . . . . . . . . . . . . . . 113
6.3.5 Experiment Two . . . . . . . . . . . . . . . . . . . . . . . . 114
6.3.6 Experiment Three . . . . . . . . . . . . . . . . . . . . . . . 115
6.3.7 Experiment Four . . . . . . . . . . . . . . . . . . . . . . . . 116
6.3.8 Experiment Five . . . . . . . . . . . . . . . . . . . . . . . . 117
6.4 Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
6.4.1 General Trends . . . . . . . . . . . . . . . . . . . . . . . . . 118
6.4.2 Specifics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120

**7 Behavioural identification experiments** **121**

7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
7.2 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
7.3 The Data-Gathering Program . . . . . . . . . . . . . . . . . . . . . 122
7.3.1 Real-time Data . . . . . . . . . . . . . . . . . . . . . . . . . 123
7.3.2 Choice of Malicious Examples . . . . . . . . . . . . . . . . . . 123
7.3.3 Final Datasets . . . . . . . . . . . . . . . . . . . . . . . . . 124
7.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
7.4.1 Experiment One . . . . . . . . . . . . . . . . . . . . . . . . 125
7.4.2 Experiment Two . . . . . . . . . . . . . . . . . . . . . . . . 125
7.4.3 Experiment Three . . . . . . . . . . . . . . . . . . . . . . . 126
7.4.4 Experiment Four . . . . . . . . . . . . . . . . . . . . . . . . 127
7.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

**8 Conclusions and Future Work** **129**
8.1 Evaluation . . . 129
8.2 The Original hypotheses and the final contribution . . . 130
8.3 The PhD: A retrospective review . . . 131
8.4 Future Work . . . 132
8.4.1 Anomaly Detection . . . 134
8.4.2 Combining Anomaly Detection and Authentication . . . 134
8.4.3 A Network for Malware Research . . . 135
8.4.4 A Complete Protection System . . . 136
8.5 Final Thoughts . . . 138

**Appendix A Detailed Structural Results** **140**
A.1 Experiment One: detecting specific categories of malware . . . 141
A.1.1 Virus-NonMal . . . 141
A.1.2 Trojan-NonMal . . . 142
A.1.3 Backdoor-NonMal . . . 143
A.1.4 Worm-NonMal . . . 144
A.2 Experiment Two: pairwise malware comparisons . . . 145
A.2.1 Backdoor-Worm . . . 145
A.2.2 Trojan-Backdoor . . . 147
A.2.3 Trojan-Worm . . . 148
A.2.4 Virus-Backdoor . . . 149
A.2.5 Virus-Trojan . . . 150
A.2.6 Virus-Worm . . . 151
A.3 Experiment Three: three-way comparisons of malware . . . 152
A.3.1 Backdoor-Trojan-Worm . . . 152
A.3.2 Backdoor-Virus-Trojan . . . 153
A.3.3 Backdoor-Virus-Worm . . . 154
A.3.4 Trojan-Virus-Worm . . . 155
A.4 Experiment Four: four-way comparison of malware . . . 156
A.4.1 Backdoor-Trojan-Virus-Worm . . . 156
A.5 Experiment Five: non-malware versus malware . . . 157
A.5.1 Part 1: Backdoor-Trojan-Virus-Worm-NonMal . . . 157
A.5.2 Part 2: Mal-NonMal . . . 158
A.6 Summary . . . 159

**Appendix B Detailed Behavioural Results** **160**

B.1 Experiment One . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161

B.1.1 Backdoor-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . 161

B.1.2 Trojan-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . 162

B.1.3 Worm-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . 163

B.2 Experiment Two . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164

B.2.1 Backdoor-Worm . . . . . . . . . . . . . . . . . . . . . . . . . 164

B.2.2 Trojan-Backdoor . . . . . . . . . . . . . . . . . . . . . . . . 165

B.2.3 Trojan-Worm . . . . . . . . . . . . . . . . . . . . . . . . . . 166

B.3 Experiment Three . . . . . . . . . . . . . . . . . . . . . . . . . . . 167

B.3.1 Backdoor-Trojan-Worm . . . . . . . . . . . . . . . . . . . . 167

B.3.2 Backdoor-Worm-Nonmal . . . . . . . . . . . . . . . . . . . . 168

B.3.3 Trojan-Backdoor-Nonmal . . . . . . . . . . . . . . . . . . . 169

B.3.4 Trojan-Worm-Nonmal . . . . . . . . . . . . . . . . . . . . . 170

B.4 Experiment Four . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171

B.4.1 Part One: Backdoor-Trojan-Worm-Nonmal . . . . . . . . . . . 171

B.4.2 Part Two: Mal-Nonmal . . . . . . . . . . . . . . . . . . . . . 172

B.5 General Trends . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172

**Appendix C The Data Gathering Program** **173**

C.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173

C.2 General Design and User Interface . . . . . . . . . . . . . . . . . . . 173

C.3 Collecting Structural Data . . . . . . . . . . . . . . . . . . . . . . . 174

C.4 Collecting Behavioural Data . . . . . . . . . . . . . . . . . . . . . . 175

C.5 Data Processing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 177

C.6 Acknowledgements . . . . . . . . . . . . . . . . . . . . . . . . . . . 177

**Bibliography** **179**

**Index** **188**

# List of Figures


2.1 How Different File Infectors Infect Files . . . . . . . . . . . . . . . . . . . 23

2.2 A diagram of the process by which spammers use zombie (virus-infected) computers to send spam. Image and description below copyright ©Foobar Obfusco 2005, licensed under the GNU Free Documentation License) . . . . 34

3.1 The sequence of attack steps described as a flow chart. . . . . . . . . . 60

3.2 The first few nodes of the probability tree. Terminal nodes at which access has been gained are shaded (red). Only a few nodes beyond level three are shown. . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

5.1 The Asylum backdoor client program by 'Slim' . . . . . . . . . . . . . . 89

6.1 Data Gathering Process . . . . . . . . . . . . . . . . . . . . . . . . . . 107

6.2 Learning curves for Mal-NonMal (Top: J48; Bottom: Naive Bayes). Experiment 5(2), section A.5.2. . . . . . . . . . . . . . . . . . . . . . . 112

6.3 Learning curves for Virus-NonMal (Top: J48; Bottom: Naive Bayes). Experiment 1, section A.1.1. . . . . . . . . . . . . . . . . . . . . . . . 112

6.4 Structural Experiment One . . . . . . . . . . . . . . . . . . . . . . . 114

6.5 Structural Experiment Two . . . . . . . . . . . . . . . . . . . . . . . 114

6.6 Structural Experiment Three . . . . . . . . . . . . . . . . . . . . . . 115

6.7 Structural Experiment Four . . . . . . . . . . . . . . . . . . . . . . . 116

6.8 Structural Experiment Five . . . . . . . . . . . . . . . . . . . . . . . 117

7.1 Behavioural Experiment One . . . . . . . . . . . . . . . . . . . . . . 125

7.2 Behavioural Experiment Two . . . . . . . . . . . . . . . . . . . . . . 126

7.3 Behavioural Experiment Three . . . . . . . . . . . . . . . . . . . . . 126

7.4 Behavioural Experiment Four . . . . . . . . . . . . . . . . . . . . . . 127

8.1 Example Network for Malware Research . . . . . . . . . . . . . . . . . . . 135

8.2 Data collection and detection of infection. Here any appropriate adaptive pattern recognition program could be used. (Process boxes blue, Data boxes green.) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136

8.3 System Block Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

C.1 The initial VMON program window . . . . . . . . . . . . . . . . . . . . 173

C.2 Diagram of Key Classes in the VMON Program . . . . . . . . . . . . . 174

C.3 The ProgramDataCapture dialog box . . . . . . . . . . . . . . . . . . . 175

C.4 The ProcessDataCapture Dialog Box . . . . . . . . . . . . . . . . . . . 175

C.5 The ProcessSystemQuery Dialog Box . . . . . . . . . . . . . . . . . . . 176

C.6 Diagram of Behavioural Data Storage Object . . . . . . . . . . . . . . . 177

C.7 The About Box, containing the Mersenne Twister copyright notice . . . 178

# List of Tables

2.1 Salomon's Seven Types of Malware . . . . . . . . . . . . . . . . . . . . 22

2.2 Some of the structural anomalies common in infected program files given by Szor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.1 Probabilities $P(10, k, d)$ for the overlap . . . . . . . . . . . . . . . . . 61

4.1 Example C4.5 Input Data Files . . . . . . . . . . . . . . . . . . . . . . 72

4.2 Example WEKA ARFF File . . . . . . . . . . . . . . . . . . . . . . . . 74

5.1 Some of the structural anomalies common in virus-infected program files as described by Szor . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.2 Diagram of PE File Structure . . . . . . . . . . . . . . . . . . . . . . . 79

5.3 Call-to-Pop: Simplest form . . . . . . . . . . . . . . . . . . . . . . . . . 84

5.4 Call-to-Pop in AlephOne's Shellcode . . . . . . . . . . . . . . . . . . . 85

5.5 Call-to-Pop: Covert . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

5.6 Call-to-Pop in use . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

5.7 Some of the behavioural properties that might be expected to help detect malware. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

5.8 The commands used by Asylum's communication protocol . . . . . . . 91

5.9 Excerpt from Welchia's unpacking code . . . . . . . . . . . . . . . . . . 93

5.10 Registry Access Code from Backdoor.Win32.Asylum . . . . . . . . . . 96

5.11 Contents of a PROCESS_MEMORY_COUNTERS structure . . . . . . 101

6.1 Structural Attributes Used for Classification . . . . . . . . . . . . . . . 108

6.2 Baseline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

6.3 Structural Experiment Four: J4.8 Confusion Matrix . . . . . . . . . . . 117

6.4 Structural Experiment Five (Part One): J4.8 Confusion Matrix . . . . 118

7.1 Process Attributes Sampled Over Time . . . . . . . . . . . . . . . . . . 123
7.2 Malware used for behavioural data capture . . . . . . . . . . . . . . . . 124

A.1 Results from WEKA Classifiers Applied to Virus-NonMal . . . . . . . . 141
A.2 Results from WEKA Classifiers Applied to Trojan-NonMal . . . . . . . 142
A.3 Results from WEKA Classifiers Applied to Backdoor-NonMal . . . . . 143
A.4 Results from WEKA Classifiers Applied to Worm-NonMal . . . . . . . 144
A.5 Results from WEKA Classifiers Applied to Backdoor-Worm . . . . . . 145
A.6 Results from WEKA Classifiers Applied to Trojan-Backdoor . . . . . . 147
A.7 Results from WEKA Classifiers Applied to Trojan-Worm . . . . . . . . 148
A.8 Results from WEKA Classifiers Applied to Virus-Backdoor . . . . . . . 149
A.9 Results from WEKA Classifiers Applied to Virus-Trojan . . . . . . . . 150
A.10 Results from WEKA Classifiers Applied to Virus-Worm . . . . . . . . . 151
A.11 Results from WEKA Classifiers Applied to Backdoor-Trojan-Worm . . 152
A.12 Results from WEKA Classifiers Applied to Backdoor-Virus-Trojan . . . 153
A.13 Results from WEKA Classifiers Applied to Backdoor-Virus-Worm . . . 154
A.14 Results from WEKA Classifiers Applied to Trojan-Virus-Worm . . . . 155
A.15 Results from WEKA Classifiers Applied to Backdoor-Trojan-Virus-Worm 156
A.16 Results from WEKA Classifiers Applied to Experiment 5 (part 1) . . . 157
A.17 Results from WEKA Classifiers Applied to Experiment 5 (part 2) . . . 158
A.18 Summary: Classifier Accuracy on Test Data . . . . . . . . . . . . . . . 159

B.1 Backdoor-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
B.2 Backdoor-Nonmal: Results by Class . . . . . . . . . . . . . . . . . . . 161
B.3 Trojan-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
B.4 Trojan-Nonmal: Results by Class . . . . . . . . . . . . . . . . . . . . 162
B.5 Worm-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
B.6 Worm-Nonmal: Results by Class . . . . . . . . . . . . . . . . . . . . 163
B.7 Backdoor-Worm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
B.8 Backdoor-Worm: Results by Class . . . . . . . . . . . . . . . . . . . 164
B.9 Trojan-Backdoor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
B.10 Trojan-Backdoor: Results by Class . . . . . . . . . . . . . . . . . . . 165
B.11 Trojan-Worm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166

B.12 Trojan-Worm: Results by Class . . . . . . . . . . . . . . . . . . . . . . 166
B.13 Backdoor-Trojan-Worm . . . . . . . . . . . . . . . . . . . . . . . . . . 167
B.14 Backdoor-Trojan-Worm: Results by Class . . . . . . . . . . . . . . . . 167
B.15 Backdoor-Worm-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . 168
B.16 Backdoor-Worm-Nonmal: Results by Class . . . . . . . . . . . . . . . . 168
B.17 Trojan-Backdoor-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . 169
B.18 Trojan-Backdoor-Nonmal: Results by Class . . . . . . . . . . . . . . . 169
B.19 Trojan-Worm-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . . 170
B.20 Trojan-Worm-Nonmal: Results by Class . . . . . . . . . . . . . . . . . 170
B.21 Backdoor-Trojan-Worm-Nonmal . . . . . . . . . . . . . . . . . . . . . 171
B.22 Backdoor-Trojan-Worm-Nonmal: Results by Class . . . . . . . . . . . . 171
B.23 Mal-Nonmal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
B.24 Mal-Nonmal: Results by Class . . . . . . . . . . . . . . . . . . . . . . 172

Chapter 1

# Introduction

## 1.1 What Is Malicious Software?

A **malicious program** is a program *deliberately* designed to cause some form of harm or damage to the user or the system. Examples would be viruses, Trojan horses, backdoors, and spyware, all of which are referred to by the umbrella term 'malware'.[1] The problem of malicious software (or 'software written for malicious purposes') will be familiar to most people – by now, nearly any computer user will have had some experience of it. An increased number of online transactions means most malicious software is now written and deployed by fraudsters or organised criminals [Bradbury, 2006]. Malicious software can be used to send untraceable spam email, to steal confidential data, or as a means of extortion. Existing approaches to control of malicious software have limitations, and research into possible new methods is therefore worthwhile.

As the interconnection of computer systems has risen the need for intelligent flexible counter intrusion systems has become imperative. Existing systems, such as signature-based malware (virus) scanners and Intrusion Detection Systems, have disadvantages. For example, malware scanners will only function properly if they are constantly updated with new malware signatures. Software companies must therefore continue to produce updated signature files. If they do not charge for this service they may lose money, but if they do, users are less inclined to update. Current Intrusion Detection Systems are predominantly concerned with protection against attacks launched from

[1]This definition intentionally excludes cases of software which causes unintentional harm (e.g. owing to a bug), and will also leave aside philosophical questions about the intent of the author or introducer of the software (who might be termed a "malicious agent").

outside via network services. For total protection, it would be ideal to have a system which could combine protection against attackers on the outside with protection against malicious software from the inside.

## 1.2 The Connectivity Issue

Before computers were networked on a large scale, connectivity was not really an issue, as the only way most people could transfer data between computers was by moving disks or tapes from one machine to another. Someone who wished to gain unauthorised access to a computer would first need to break in to the building and room in which it was situated, and someone who wished to introduce malicious software would have to have the same physical access to the data media (disks or tapes) or the computer. Computer security, in those days, was mostly a matter of physically securing the computer and its media.

At the present time, however, almost every computer is networked in some way – even laptops and mobile devices (PDAs or mobile telephones) are generally connected. In 1995 Internet Protocol version 6 [Deering and Hinden, 1995] was introduced to serve the growing demand for network addresses – and there was much talk of a time when all consumer appliances – fridges, freezers, toasters, washing machines, televisions – would have an internet connection. That point may not have been reached in practice, but the number of consumer appliances with network connectivity is growing. Witness the current popularity of 'media centres' – specialised computers which perform all the functions of a DVD recorder, set-top box, games console and music centre. Microsoft's XBox 360, though primarily a games console, is in reality an example of this – its website proudly states that users can "Play the most compelling games. Watch DVD movies. Enjoy digital music, photos, and videos in an integrated entertainment system." [Microsoft, 2005].

Unfortunately, such widespread connectivity raises many security issues which may not have been anticipated, since most of the impetus for consumer appliance connectivity has come for other reasons. It is conceivable that appliance-cracking may become as popular as computer-cracking. Despite the obvious objection that the only people who would get a kick out of making people's food go off or burn or replacing their video libraries with offensive or pornographic material are immature e-vandals, more serious scenarios are possible – for example, burglars being able to compromise and disable alarms and access control systems if these are connected to the Internet. If,

as predicted, there is a rapid increase in connected consumer devices, then an equally rapid increase in security awareness will be needed if a security disaster is to be avoided. At the moment home users have enough trouble keeping their computers secure and free of malicious software: the task will become much more complicated when all their appliances are online as well.

The subject of 'information warfare' was widely discussed during the 1990s. An oft-cited early commentator on this issue was Winn Schwartau, whose book on the subject [Schwartau, 1994] provides extensive analysis with reference to the global sociopolitical and economic conditions of the early 1990s. While it is true that the more extreme scenarios it outlines have not come to pass, many of the smaller predictions it makes have been accurate, including the prediction of the rise of international terrorism. Schwartau makes the point that information has become the most valuable commodity in the modern economy. Anyone who can disrupt, corrupt, or destroy information has a potent weapon. Despite over ten years having passed since Schwartau's book, one of his main points – that most organisations give little thought to protecting their information assets – remains true in many cases. Some factors have changed since Schwartau wrote his book. For instance, passive eavesdropping has become much easier owing to the introduction of wireless networking. In Schwartau's scenarios, would-be eavesdroppers had to rely on the use of Van Eck-type equipment to pick up stray radiation from display units [Eck, 1985], or electronic snoopers which were physically connected to network wiring. As wired networks are replaced by wireless ones, these eavesdropping methods from the past become obsolete: all that is needed now is a laptop with a wireless network card and the appropriate software. Early encryption methods such as WEP were repeatedly shown to be insecure [Cam-Winget *et al.*, 2003]. Even now that better encryption and authentication is available for wireless networks, users (even businesses) resist deploying it because of the inconvenience it entails – particularly in an organisation which has many visitors who require legitimate access to the wireless network.

A company may pride itself on its commitment to security, its constantly-updated firewall, committed system administrators, VPNs, Intrusion Detection Systems, etc., but all this impressive security technology is rendered instantly useless if the internal network traffic passes unencrypted across the company wireless LAN and can be picked up by someone across the street with a laptop. In large cities, where many companies have wireless LANs, the sport of 'wardriving' – moving around an urban area with a wireless-enabled computer trying to find and gain access to wireless networks – has proliferated. The main motivation for wardriving is to obtain free internet access:

however, the potential for security breach should be obvious.

Even if nothing is compromised on the internal network, unwittingly sharing one's internet access with the rest of the world is not a good idea. Assume someone is able to gain internet access via an unsecured business WLAN. He or she could then proceed to malicious or illegal activities – hacking, viewing child pornography, blackmail, fraud, etc. – and any tracing would lead back to the company whose LAN was abused, who would then have a difficult time proving that it was *not* one of their employees who committed the crime! In any case, the genuine perpetrator is highly unlikely to be caught. Back in the days before widespread computer networking and when telephone switching was still analogue, persons known as "phone phreaks" or "telephone hackers" would explore telecommunications networks in much the same way that their later counterparts did computer networks. Telephone hackers who wished to perform high-risk exploration without being traced would take steps to obscure the point at which they were accessing the telephone network. The simplest way to do this was to gain access to someone else's telephone line, usually via a telephone company junction box on the outside of a house or in the street. A payphone could also be used, especially in the USA where local calls were often free of charge (telephone hackers had many ways of fooling switching equipment into connecting other types of call at local rates). This way, when the telephone company and/or law enforcement finally tracked down the source of all the fraudulent calls, they ended up arresting the innocent owner of the compromised telephone line. The same techniques were later used by the first computer hackers to avoid paying for their extensive data calls and to further frustrate tracing attempts by law enforcement. Wardriving could be seen as the 21st-century equivalent, with the added bonuses that you are less likely to get caught (no wires) and it can be done in relative safety from the comfort of your vehicle.

Another of Schwartau's points is that malicious software programs may be used as weapons in information warfare. During the first Gulf War, it was rumoured that the National Security Agency had embedded a virus in a chip that controlled a certain printer, one or more of which were subsequently sold to Iraq. During the war, the virus was activated and used to destroy the Iraqi air defence system. Despite the small matter of the source material being dated April 1st, not to mention the many technical problems such a method would face, national news agencies in America are said to have broadcast the story as fact. With present-day technology, an attack based on a similar method could certainly work in many circumstances. Despite increased awareness of the need for information security since 1994, many laypeople still often assume that information security is only about preventing hackers from accessing your

system from outside over a network. Conventional Intrusion Detection Systems (IDS) concentrate on protecting against this threat. The state of technology to defend against malicious programs – which may be introduced unwittingly by legitimate users inside an organisation – is much less advanced. Though various alternatives have been proposed, the basic signature-based virus scanner still reigns supreme.[2] Few higher technology alternatives exist outside classified military research – Schwartau reports rumours that governments and military organisations have been interested in the offensive potential of computer viruses for some time. If this is so, it would be logical for them also to be considering defensive measures against such viruses. One factor which must constantly be considered when investigating malware is the principle of 'greatest cost to the user'. In other words, there are certain types of malware which have the potential to produce the 'worst-case scenario', and both the scenario and the malware have changed. Back in the early days, losing data was the worst thing that could happen. Whilst loss of data would undoubtedly still be a major concern, an argument could be made that it is no longer the most serious thing that could happen in the event of a malware 'attack'. As it is, in today's networked world, the strongest contender is undoubtedly the leaking of private or confidential information. Since online shopping and banking became popular, more and more individuals and businesses are storing confidential information such as bank and credit card details on networked personal computers. Countermeasures are sometimes taken by banks, but in some cases these have been shown to be ineffective [Goring *et al.*, 2007], and consequently, malware which is specifically designed to acquire such information and feed it back to the author or controller is going to become a serious problem.

The risk that an average individual internet user will suffer a Distributed Denial of Service (DDoS) attack is quite small. The risk for a business is a little larger, particularly if they are large or have much 'web presence'. However, the cost of a DDoS attack is insignificant compared to the cost of having your customers' credit card details leaked [Thurston, 2007]. It is well known that many DDoS and cracking attacks are now being orchestrated by organised crime interests for the purposes of extortion [Bradbury, 2006]. It is also to be expected that criminals will start attempting to obtain confidential information via specifically targeted malware.

In a conventional DDoS attack it is necessary for the controller of the botnet to send some kind of signal to start the attack – possibly leading to their detection. Similarly, malware which steals data generally has to know how to send it back to the controller,

[2]This is a slight assumption, since manufacturers tend not to release much information on their products, but it is certainly true that all existing anti-malware products require regular updates.

who can thus be traced. However, this could be avoided via the use of *covert channels*. Consider a hypothetical situation where a piece of malware, propagated via virus or worm techniques and covertly present on a home user's PC, could leak confidential information in the form of a (perhaps steganographically-disguised) newsgroup posting. Most newsgroup users would ignore the message – but someone 'in the know' could read between the lines and extract the relevant information.

In many countries it is a criminal offence to gain unauthorised access to a computer system, whether to extract confidential information or otherwise: however, if a computer, under the control of a piece of malware, sends confidential information to the public domain, the legal position is trickier: the author of the malware would almost certainly still be liable, but it may be impossible to trace them, and it would certainly be impossible to arrest everyone who read the newsgroup. All in all, it is arguable that the need for protection against malware has never been greater.

## 1.3 Combat Approaches

A full survey of the approaches that are currently used to combat malware will be given in section 2.4 of Chapter 2. However, it is possible (with some overlap) to divide past and present approaches to malware into four main categories:

- **Detection**: the determination that a given program is likely to be malicious.
- **Identification**: classifying a given program as identical to or related to a known piece of malware.
- **Prevention**: stopping a given program from carrying out hostile functions.
- **Recovery**: removing a malicious program from a system and/or repairing any damage it has caused.

An ideal system to protect against malware ought to span all these categories, though the most popular practical systems at the present time are category 2 and 4 (signature-based malware scanners). Academic research has proposed various solutions which fit categories 1 and 3, but these have rarely resulted in practical implementations. For the purpose of this thesis, consideration will only be given to the first three categories.

## 1.4 Scope and Aim

### 1.4.1 Hypotheses

1. An automated classification system, provided with structural data on malicious and non-malicious programs, should be able to distinguish between the two classes with a high degree of accuracy.

2. Since other types of malicious program, such as Trojans, do not necessarily have marked structural anomalies, one might expect that a structural classification system would have much more difficulty distinguishing them from non-malicious programs.

3. In contrast an automated classification system, provided with *behavioural data* instead of structural data, should be able to classify all types of malicious software with equal success.

### 1.4.2 Aims

The aim of this PhD, broadly stated, is to determine whether the detection of malicious programs can be automated using a learning system, and to develop programs utilising this concept. Such programs might eventually form parts of an intelligent, flexible counter-intrusion system that can defend a computer or network against malicious and unauthorised programs. Use will be made of an isolated computer on which viruses and other malicious programs may run, in order that their characteristics can be studied safely. Software tools to analyse what is running on the isolated machine will be developed. These tools will gather pertinent statistics and output results which can be fed into learning systems. Experiments can then be performed to determine whether machine learning systems can be used to detect malicious software, firstly by determining whether the rules produced by the learning systems from a set of training data correspond to the rules already produced by human experts, and then by determining whether the learning systems can classify unseen programs as benign or malicious with a high degree of accuracy.

If this proves practical, the ideas could be used to develop some or all of the components of an intelligent and flexible counter-intrusion system that can defend a computer network against malware. By "intelligent and flexible" it is meant that the response of the counter-intrusion system should be reasonably context-sensitive (e.g. the response

will differ in different circumstances). However, the development of such a system is far beyond the scope of a PhD thesis.

### 1.4.3 Principal Contributions

The major contribution of this thesis is to show that the use of such systems could help bridge the vital time-gap, which exists at present, between the central detection of a new outbreak and the manual creation of a new signature file for download to subscribers' computers. In essence, the thesis demonstrates that the decision tree classifier approach is perfectly feasible on simple structural data and offers a relatively high degree of protection. Whilst a preliminary investigation for the same techniques applied to behavioural data has also been conducted, the results were less clear-cut.

The structure of the thesis is as follows: Chapter 2 will give a brief historical overview of the development of different types of malware and malware combat methods, followed by a review of the academic literature on the subject. Chapter 3 is a case study which uses a real-life security flaw to illustrate both the actual and potential uses of malware for financial fraud. Chapter 4 gives information about the machine learning systems and classification methods chosen and the reasons for their choice. Chapter 5 describes how structural and behavioural program/process data may be collected on the Windows operating system.[3] Chapter 6 gives information on the structural classification experiments, with analysis of the results, and Chapter 7 gives similar information on the behavioural classification experiments. Finally, Chapter 8 gives an overall evaluation of the results and a retrospective on the PhD project, and lists future work that could be done in this subject area.

Any trademarks referred to in this thesis are the property of their respective owners.

[3]Investigation was restricted to the Windows operating system due to the vast majority of malware being written for Windows.

# Chapter 2

# Historical Background and Literature Review

This chapter starts with an overview of the different types of malware and their history, and the development of tools to combat them. This is followed by a review of the available literature.

## 2.1 Types of Malware

This section aims to give an overview of the more common types of malicious program. The definition of a malicious program ('malware') varies. Authors such as [Gollmann, 1999] gives only four types: Trojan horses, viruses (resident and transient), logic bombs and worms. Confusingly, Gollmann seems to refer to all malware as 'viruses' (see for instance [Gollmann, 1999] pp134–136). On the other hand, David Salomon gives seven types ([Salomon, 2006] p34). Salomon's list is reproduced here in Table 2.1, since it includes distinctions not made by other authors:

| Type | Description |
|---|---|
| Virus | Resides in an executable file and propagates to other executables |
| Logic bomb | A virus whose payload is delayed and is triggered by some event in the computer |
| Time bomb | A special case of logic bomb where the trigger is a particular time or date |
| Rabbit | A virus whose payload is to annoy and vex the user rather than destroy data |
| Backdoor | A hidden feature (normally in Trojans or spyware) that gives certain people special privileges |
| Worm | Executes independently of other programs, replicates itself, and spreads through a network |
| Trojan horse | Hides in the computer as an independent program and has a malicious function |

Table 2.1: Salomon's Seven Types of Malware

Salomon's list is probably the most comprehensive available in the literature, and he also recognises that some of the definitions are overlapped (for instance Backdoors and Trojans are not always distinct).

### 2.1.1 Viruses

Salomon's definition of a virus has already been given in Table 2.1. However, a more general definition is given by Bishop: "a sequence of instructions that copies itself into other programs in such a way that executing the program also executes that sequence of instructions." [Bishop, 1992]. Like biological viruses, true computer viruses *cannot exist independently from their host programs.* Unfortunately the term "virus" is often used as a generic term for any piece of malware,[1] which means that the reader may encounter this sense even here, especially in quotes or references to other authors.

There are many types of virus. This section will give a relatively simple overview of the more common types. More technical details will be given in subsequent chapters. The reader is also referred to Peter Szor's monumental "The Art of Computer Virus Research and Defense" [Szor, 2005], one of the most up-to-date books on the subject.

Figure 2.1 shows how three types of file-infecting virus modify program files.

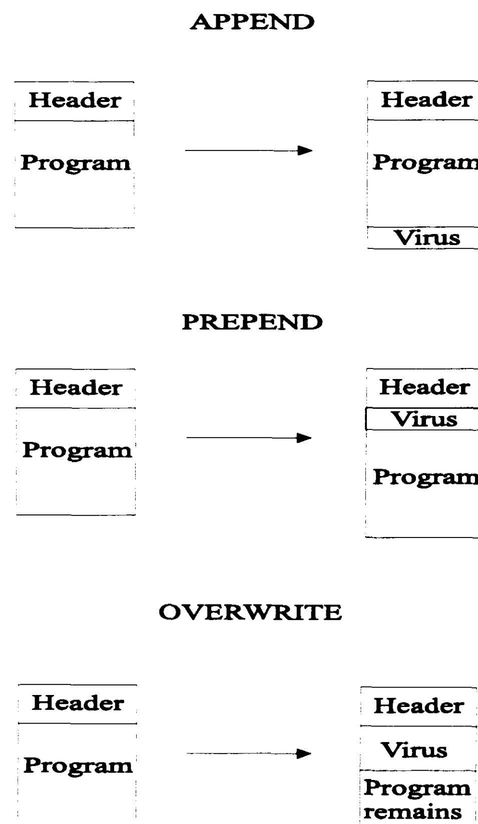



Figure 2.1: How Different File Infectors Infect Files

[1]Furthermore, common terms for malware-fighting software are 'antivirus' or 'virus scanner' rather than 'anti-malware' or 'malware scanner'.

### Overwriting Viruses

These are the simplest viruses. As their name suggests, these viruses overwrite target programs in such a way that the program is no longer able to carry out its original function. The older overwriting viruses could generally only cope with simple executable file formats (such as the DOS COM format, inherited from CP/M). One advantage of overwriting is that the virus cannot be detected by simply observing file size changes. Furthermore, such viruses are very easy to write, even in high-level languages on modern operating systems with complicated file formats. They thus represent very little of a technical challenge for virus writers, and since such viruses are very easy to detect as infected programs no longer work, they remain uncommon.

### Appending and Prepending Viruses

The viral code is appended or prepended to the target executable, and the executable header is patched so that the viral code executes first, then passing control back to the target. If the viral code is sufficiently small and efficient, the user of the target program will be unaware of any change in its function. Appending viruses that attack complex executable files (e.g. Win32 PE format, UNIX ELF/COFF formats) generally append a new section to the file, altering the file headers, but sometimes they can make use of unused areas in existing sections, or enlarge the sections by patching the section table.

Prependers can be as complex as appenders, but are often far simpler, as illustrated by the variety which merely creates a new copy of itself with the target executable compressed and stored as data within the executable. Then, the target executable is deleted and the new virus executable is given the target's original name. When the new executable runs, the virus code executes first, then the code for the target executable is decompressed, copied to a temporary file, and executed. This type of prepender virus might be expected to appeal to novice virus writers as it can be written in a variety of high-level languages ([Szor, 2005] p135) and requires little in the way of technical skill.

### Memory-resident Viruses

A memory-resident virus is a virus which remains in memory and continues to infect files rather than infecting just a few files and then terminating. This sort of virus was common in the days of MS-DOS, because the service routines for various interrupts – used by DOS as an API – could be modified by any program. It was thus possible for

a virus to load itself into memory and redirect one or more interrupt service routines to point to its own code, then terminate itself without freeing the memory. Whenever a hooked interrupt was called, the virus would be executed.[2] Viruses used the hooked interrupts to locate files to infect, as well as to hide the results of infections (such as increased file sizes).

The most well-known form of memory-resident virus is the boot sector virus, which is covered below. Boot sector viruses generally used interrupt hooks to determine when a disk was inserted or read from. They would then write themselves to its boot sector, thus infecting the new disk. Like the boot-sector virus, the form of resident virus described here died out with MS-DOS (e.g. Windows 98/Me), but equivalents were soon developed, and for a full description of these the reader is directed to Chapter 5 of [Szor, 2005].

### Boot Sector Viruses

Boot sector viruses are a special case of memory-resident virus. The 'boot sector' of a disk contains the code which starts the operating system. When a computer is switched on, the BIOS, having completed its various initial functions, passes control to the program in the boot sector. All disks contain a boot sector even if they are not actually bootable: therefore, in the days when floppy disks were more widely used to transfer data between computers, boot sector viruses were extremely common. Generally, these viruses infected the boot sector of the hard drive, which meant that they were loaded whenever the computer was started. They then subverted the disk access routines of the operating system (usually DOS, in which case the subversion was done by hooking Interrupt 19H) so that they could infect any floppies that were used in the machine.

The Michaelangelo virus was a fairly run-of-the-mill boot sector virus which had a destructive payload [Kephart *et al.*, 1993]. If an infected computer was booted on March 6th, the virus would destroy all the data on the hard drives. Thus, the weeks before March 6th 1992 were characterised by doom-laden press reports and hysterics from computer users. However, as in most cases, the situation was severely over-hyped: IBM estimated that more hard drives died of routine hardware failure on March 6th 1992 than were affected by the virus [Kephart *et al.*, 1993]. The massive epidemic that the press had been predicting did not occur – however, massive public demand

[2]It should be noted that this idea, known as Terminate-Stay-Resident (TSR), was also used by legitimate programs.

for antivirus products was generated, with some vendors selling more of their product in the week before March 6th than in all of the rest of the year.

Boot sector viruses could spread rapidly but most did not contain destructive payloads – the Michaelangelo virus was a notable exception. Others, such as the ANTICMOS family, contained a payload which, as the name suggests, attempted to corrupt the target's BIOS – though it is unclear whether this ever succeeded. Boot sector viruses are generally thought to be obsolete (along with floppy disks) but with the advent of bootable CDs and DVDs and Flash RAM disks, there is clearly some potential for new boot sector viruses to be developed.

### Macro Viruses

Macros are defined by Wikipedia as "an abstraction, whereby a certain textual pattern is replaced according to a defined set of rules." Macros are often used in compilers and assemblers to allow programmers to shortcut frequently-used sequences. Eventually, the term was applied to any software which gave the user the facility to automate tasks – present in many programs since the early days of personal computing. Most of these originally used a 'record/playback' system – users would set the program to 'record' their input, and do a task manually. They could then 'play back' that task whenever they wanted to do the same thing (the Recorder in Windows 3.0 was an early example). Later, programming languages specifically designed for writing macros were produced.

Many products produced by Microsoft contain inbuilt support for macros written in a special form of the BASIC programming language known as Visual Basic for Applications (or VBA). VBA can perform many functions normally restricted to 'real' programming languages (e.g. reading and writing files) – and was also designed to be compatible across the entire Microsoft Office suite. As a result it was not long before the first macro viruses appeared in Microsoft Word documents. When an infected document was opened, the macro code within it would run automatically, installing itself in the user's 'document template' file. This meant that every new document produced by that user would be infected. Microsoft did provide a warning message about macros, but most users ignored it or switched it off, especially if they used macros for legitimate purposes. Macro languages have also been used to create email worms.

### 2.1.2 Worms

Worms differ from viruses in that they are autonomous programs in their own right. Their aim is to spread from computer to computer, and they usually exploit vulnerabilities in network interface software to do this. When the worm runs, it connects to a remote computer and attempts to inject its exploit code via a known vulnerability. The worm's exploit code will attempt to subvert the vulnerable process into somehow receiving a copy of the worm's code, saving it to a file and executing it on the target computer.

In 1988, Robert Morris, a graduate student at Cornell University and, ironically, the son of the head of the NSA's Computer Security division, developed what came to be known as the Internet Worm. Morris' program, which he claimed had escaped accidentally, spread rapidly across the Internet (which was of course vastly smaller than it is today) and caused widespread disruption. This event is also documented in [Stoll, 1989]. The Internet Worm spread from computer to computer by exploiting buffer overflow vulnerabilities in privileged network utility programs running on BSD Unix systems. Luckily, other operating systems were unaffected, leading Stoll to make comments about operating system diversity being a good thing. These comments seem prescient considering the Windows worms that emerged ten years later: Blaster, Sasser and all the others once again exploited buffer overflow vulnerabilities in privileged network service programs. But whereas in 1988 the various system administrators who ran the large computers which made up the Internet could neutralise the worm within a few days (according to [Stoll, 1989]), the Windows worms had a vastly larger Internet to spread around and a huge number of unpatched home computers to infect. As a result, cleaning up the mess was much harder.

#### Email Worms

As previously mentioned, VBA macros were designed to be compatible across many different Microsoft applications. This included the Microsoft email client, Outlook, or the cut-down version Outlook Express. Macros within email messages could send email, read data from address books, download and execute programs, and do many other things – all without the knowledge or consent of the user. In some cases, an infected message would not even need to be opened. Furthermore, Outlook left macros turned on by default, and not everyone knew how to turn them off. It had been common to send malicious programs as email attachments for some time – usually with something to entice the user to execute the attachment (such as a message body stating that it

contained free pornography). Now, with the addition of a suitable macro, there was no need to spend time enticing the user – they could be infected at the click of a mouse.

The first email macro worms (such as the 'Love Bug' of 2000) tended to do little damage beyond clogging up inboxes. Thanks to increased awareness and Microsoft making its Outlook email client more secure, emailed macro worms have now mostly been superseded by binary email worms (as opposed to macro-based ones). Such binary email worms usually attempt to connect to a mail server and send out multiple copies of an email with the worm attached, bypassing the user's mail client entirely. Worms may also spread using alternative communications protocols such as IRC and ICQ.

### 2.1.3 Other Email Tricks

Emailed malware employs various ruses to trick the user into running the executable payload, usually by representing it as an 'important document' or as a pornographic picture. Some such ruses are technical as well as psychological. Knowledgeable users know that one should not run executable attachments unless from a trusted source, so the viruses have started disguising the attachments as innocuous files. There is a particularly insidious way of doing this which exploits two Windows features. By default, Windows will hide file extensions from the user if the file is a recognised type (such as a program). Thus, it is possible to have a file called 'xxxpics.jpg.scr'.[3] By default, Windows will hide the '.scr' or '.exe' extension from the user, who will only see the '.jpg' extension. If the malicious author has been clever enough, he or she will have assigned an icon to the malicious program which looks like a picture (or like the standard Windows picture icon). So the user is confronted with a file which *appears* to have a picture file extension, and has the correct picture icon. What harm could there be in opening it...? An astute user may have noticed that, with extension hiding turned on, the '.jpg' extension ought to be hidden for a real picture file, but even astute users may well overlook such an apparently insignificant detail, and it is also possible to hide the 'real' file extension by putting lots of spaces or other characters in the filename, e.g.

```
xxxslutz.jpg                              .exe
```

Since email clients often used to display only the first characters of a long filename, users may not think to check before they attempt to open the file. On the other hand,

[3]For historical reasons to do with screensavers, '.scr' is a Windows executable file extension equivalent to '.exe', though users may be unaware of this.

some modern email clients will warn users if a message has an executable attachment. However, in the right circumstances, the attacker may even be able to send a genuine picture file to accomplish a malicious purpose. For instance, in 2004 a buffer overflow in a Microsoft JPEG rendering library was discovered, which meant that it was possible for an attacker to embed malicious code in a JPEG image that would get executed when the image was viewed ([Salomon, 2006] p59).

### 2.1.4 Trojan Horses

As might be guessed from the title, these are programs which appear to have an innocuous function but which contain hidden code which does something harmful. Some have claimed that the first Trojan Horse was written by the East German hacker Karl Koch ('Hagbard') whose attacks on United States military computer systems were documented by Clifford Stoll in "The Cuckoo's Egg". This could refer to pages 55-57 which documents an attempt to install a shell script which masquerades as the login program to steal passwords – as Stoll writes,

> "The hacker's Trojan horse program collected passwords. Our visitor wanted our passwords badly enough to risk getting caught installing a program that was bound to be detected. Was this program a Trojan horse? Maybe I should call it a mockingbird: a false program that sounded like the real thing. I didn't have time to figure out the difference..." – [Stoll, 1989].

Most modern Trojans employ various enticement techniques in order to induce people to download and run them (for example, by masquerading as pornography or cracked software). They may also be 'dropped' (as a payload) by viruses or worms. It seems there is no longer any requirement for a program to pretend to perform a legitimate function in order to be called a Trojan, and the definition overlaps with 'Backdoor' to a considerable extent.

### 2.1.5 Backdoors

A backdoor was originally a hidden function in a legitimate program which allowed those privy to its secret to obtain special privileges – for instance, a cryptography program could contain a backdoor which allowed a user's private key to be recovered. In the late 1990s the discovery that part of the verification mechanism for the Microsoft

Crypto API was called 'NSAKEY' led to allegations that Microsoft had installed such a backdoor at the behest of the NSA, though this was considered extremely unlikely by security experts, not least because if the NSA had wanted covertly to compromise the Microsoft Crypto API, they would have been extremely unlikely to be so obvious about it.

Modern backdoors are more likely to be Trojans of the 'remote control' type. Two of the earliest examples were 'BackOrifice'[4] and 'SubSeven', both developed in the late 1990s, though very similar programs have been developed ever since. These programs consisted of two parts: a server, which had to be installed on the target computer, and a client, which was used by the attacker. The servers, which communicated with the clients over high network ports (BackOrifice famously used port 31337 by default), would generally modify the Windows registry to ensure that they were run (transparently to the user) every time the computer was started. Attackers had almost complete control – they could access any files, delete or modify them, display messages, open and close windows and in some cases even take control of the mouse and CD-ROM drive. The definition of a Backdoor is not discontinuous: some programs considered to be Backdoors by some antivirus researchers are classified as Trojans by others. Furthermore, some legitimate remote administration software has the potential to be used as a backdoor if improperly set up or installed without a user's knowledge.

### 2.1.6 Spyware and Adware

Spyware is the term given to a piece of software (which can be a program, part of a program, or a web browser extension) which overtly or covertly collects information about the user of a computer or program. Adware refers to advertising-supported software, but there is a substantial overlap with spyware as adware often also collects user data in order to display targeted advertisements.

Information collected by spyware or adware can range from relatively harmless usage statistics through to outright invasions of privacy. The motivation for adware and spyware is commercial – advertisers will pay large sums for the data it can gather. Sometimes, the data are used to display tailored advertisements to the user, but in other situations the user may be entirely unaware that the data gathering is going on. In the early days, programs which had spyware or adware components would mention it openly in the licence agreement, often claiming that the only way they could justify

[4]The name was a pun on a Microsoft product called BackOffice Server, which was discontinued in 1999.

releasing their program as freeware or shareware was by generating revenue through a data-gathering/targeted advertising combination. However, modern spyware is designed specifically to hide from the user (except when it is popping up advertisement windows), and to make uninstallation as difficult as possible. Such programs are often automatically downloaded and installed when a user visits a certain web site (this is known as a 'drive-by download') . Not only is this unethical, it is also unsafe, as the same techniques can and have been used by malicious programs which attempt to steal passwords etc.

The Gator eWallet program [Gator, 2005] is a particularly insidious, if dated, example of spyware. The user would download and install a program to assist them in 'remembering' passwords and form entries on web pages[5]. What the user might not have realised was that the program was also tracking their internet usage habits and sending data back to the Gator company. Furthermore, the Gator program could also be included with other software – the user might not have chosen to download and install it at all. According to [Webb, 2005] this was done in a particularly insidious manner. A user would download and install an unrelated piece of shareware, which – unknown to the user – contained a program called 'trickler'. This would then modify the user's registry so that it ran every time the computer was started. The trickler program would then slowly download the full Gator program, then install and run it. The download was done slowly so that the user wouldn't notice their bandwidth being lost. Once running, the Gator program would display advertisements while the user was online. This included replacing the original advertisement images on a web site with differing ones. Several companies sued Gator for stealing their advertising space (see [Edelman, 2002], an expert witness statement from the case, which was eventually settled in 2003).

Programs like Gator and other explicitly advertisement-supported programs had a measure of legitimacy in that users were informed, in theory at least, of the data collection process when they read the license agreement, and could choose not to use the program if they were unhappy with it. This clearly does not apply to modern examples of spyware, which are much more obviously malicious. Not all programs which collect data are spyware, but the definition is a little less clear-cut. The following lists may provide helpful criteria for distinguishing spyware and malicious adware from legitimate advertisement-supported software.

[5]This functionality was later integrated into web browsers.

**CHARACTERISTICS OF LEGITIMATE SOFTWARE**

1. The user is clearly informed that the program collects data and/or is supported by advertisements (e.g. not as a footnote hidden away at the bottom of the license agreement).
2. The user is informed exactly what data are being collected, where it is being sent, and what is being done with it (and the company or organisation receiving the data has a clearly-defined privacy policy).
3. Advertisements, if present, are displayed only within the program's main window and only when the program is in use.
4. The program does not collect any data which personally identifies the user or their computer.
5. All data-collecting components are part of the program, do not function independently of it, do not use up excess resources, and are uninstalled completely when the program is uninstalled.

**CHARACTERISTICS OF ADWARE AND SPYWARE**

1. The data collection process is hidden from the user, as is the type, quantity, destination and use of any data collected.
2. Advertisements, if present, appear as separate pop-up windows and randomly, regardless of whether the supported program is running or not.
3. The data-collection and advertisement-display features are separate and autonomous from the main program.
4. The program refuses to allow itself to be uninstalled or terminated.
5. The data-collection or advertisement-display features do not get uninstalled with the main program, but remain on the system, often going to elaborate lengths to hide themselves and prevent removal.

### 2.1.7 Botnets and DDoS Attacks

Some of the latest Trojans are designed for a specific purpose. They spread using the usual techniques (either as spammed email attachments, or as the payloads of worms or conventional viruses), or may be introduced deliberately by human crackers who have compromised a system. The Trojans, known as bots, sit unobtrusively on a computer. Each individual bot will contain code for communicating via the internet, often using the Internet Relay Chat (IRC) protocol. The malware author or controller can thus search for infected machines (which are sometimes termed 'zombies'), and send commands to them. The group of infected computers controlled by an attacker are known collectively as a botnet. Such botnets may be used for a variety of purposes, and controllers may sell or rent access to them, the commonest purchasers of such access being spammers and organised criminal groups [Bradbury, 2006].

One of the first uses for botnets was the co-ordination of denial-of-service attacks. In

such a case, whoever controls the botnet instructs each bot to launch a denial-of-service attempt against a target IP address. On its own, one bot may be insignificant, but with hundreds or thousands of zombies hammering away at an IP address, the situation is very different. Such an attack is known as a Distributed Denial-of-Service (DDoS) attack. Various companies have been subject to extortion relating to botnets, with criminals threatening to launch DDoS attacks against them unless some sort of ransom is paid. The more computers that are infected by bots, the greater the effectiveness of DDoS attacks, but the greater the risk of the bot being discovered and added to virus scanner signature lists. For this reason, modern bot authors are moving away from massive untargeted distribution via worms and towards less noticeable forms of distribution. According to a virus expert quoted by Danny Bradbury in his article on the motivation of malware authors:

> "..sending out a rapidly proliferating worm to create a huge botnet is too obvious and raises too many alarms, prompting users to take security measures. Yesterday's hobbyist malware writer was generally an adolescent male wanting to be noticed by his peers. Today's for-profit malware writers want to stay under the radar, because if their product is noticed it prompts victims to take action and reduces the number of compromised machines. This is why modern malware is less likely to deliver a payload obvious to the victim, such as deleting files from the hard drive. Organised commercial malware authors want to enslave, not destroy, their targets." [Bradbury, 2006]

Bradbury's experts also cite a second reason why bigger is not necessarily better for botnet controllers: where bots are used to steal confidential information such as credit card numbers, having a million infected hosts risks producing more data than the criminals can handle in one go, so they prefer to infect small numbers of hosts, process the data, then repeat the process. Targeted infection is usually accomplished by spamming out an infected attachment. As well as stealing confidential data, organised criminals are also using botnets as a distributed content serving system. Traditionally, those who wished to distribute illicit material via the web (e.g. child pornography or counterfeit software) or perform other illegal or objectionable activities (such as running 'phishing' pages) were vulnerable to having their servers discovered and shut down. If, however, some of the machines on the botnet are used as web servers, it becomes much harder either to remove the illicit content (because it can be replicated around the botnet) or prosecute the perpetrators (because the content is hosted on a machine belonging to someone else and without their knowledge). Sometimes, rather than storing content on infected PCs, the botnets act as proxy servers, hiding the location of the real website and allowing the real location of the content to be changed at short notice. Extortion

is also possible: some Russian Trojans will encrypt portions of a user's data, then demand payment in return for its decryption. Possibly the biggest users of botnets are the spammers. Previously spammers relied on improperly-secured web servers, which were easy to trace and fix. Now, spammers buy access to botnets, and use these to send spam. As the spam is sent from multiple locations, its source is all the more difficult to trace. The following diagram, created by a Wikipedia user, illustrates how spammers use botnets:

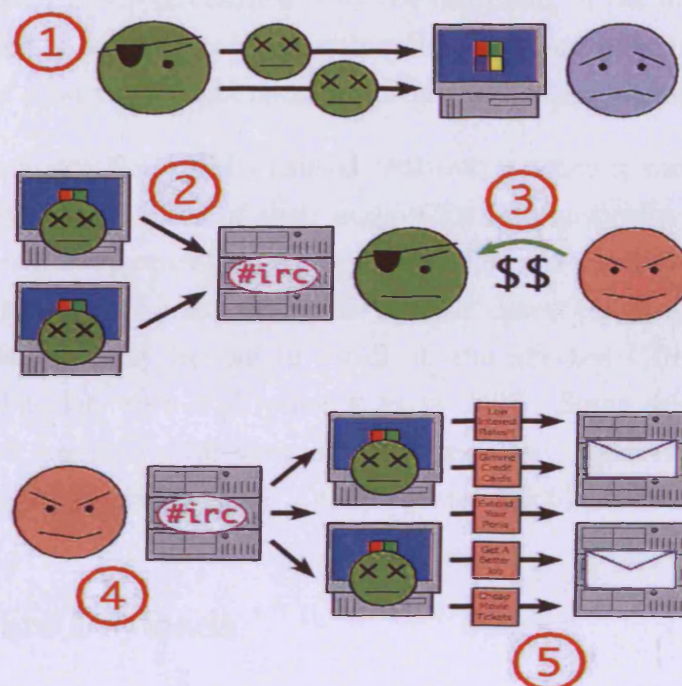



Figure 2.2: A diagram of the process by which spammers use zombie (virus-infected) computers to send spam. Image and description below copyright ©Foobar Obfusco 2005, licensed under the GNU Free Documentation License)

1. Virus (bot) writer sends out viruses (bots), infecting ordinary users' Windows PCs.

2. Infected PCs log into an IRC server or other communications medium, forming a network of infected systems known as a botnet.

3. Spammer purchases access to this botnet from virus writer or a dealer.

4. Spammer sends instructions to the botnet, instructing the infected PCs to send out spam.

5. The infected PCs send the spam messages to Internet users' mail servers.

### 2.1.8 Rootkits

Rootkits are designed to subvert the operating system at a low level. They were originally conceived for use by crackers, who would install one after successfully breaking into a system. The rootkit would allow the cracker to hide evidence of the break-in from legitimate users. Typical rootkit functions include hiding certain file types from the system, or hiding file modification dates. A typical Windows rootkit might hide all files having a certain group of characters at the beginning of the filename (i.e. 'sys'). Rootkits may work in kernel-mode (meaning that they are fully integrated into the OS) or user-mode (easier to create and install but also easier to detect).

In 2005, media company Sony BMG caused controversy when it was discovered that a copy-protection system on some of their audio CDs automatically installed a rootkit (which also doubled as spyware) on users' PCs. This was discovered accidentally by Mark Russinovich when he was testing his rootkit detection program [Russinovich, 2005], and resulted in Sony having to recall all the affected CDs and fight several lawsuits, some of which were still ongoing as of 2007. Some security experts also criticised antivirus companies for their sluggish response, given that the software in question had been in circulation since 2004 [Schneier, 2005].

### 2.1.9 Malware Payloads

The payload refers to a specific malicious function or action, generally independent of the functionality which causes the malware to spread. For instance, it is possible to take a 'benign' virus, which initially causes no damage to files other than the changes necessary for it to spread, and modify it so that it performs additional malicious actions (adding a payload). Often, the payload is triggered by a time or logical condition.[6] For instance, the Pathogen virus, extant in the early 1990s and attributed to Christopher Pile aka The Black Baron (who later became the first person to be prosecuted under UK law for disseminating malware – see below), had a highly destructive payload which would attempt to trash the hard drive of the affected computer if, among other conditions, it happened to execute on a Monday between the hours of 5 and 6pm [Probert, 1994].

[6] Some authors classify time bombs and logic bombs as specific categories of malware. However, they might more properly be considered as distinct *functionalities* which malware might have – evidently, if a time or logic bomb is to be set, it must first get itself onto a target computer, generally in the form of a backdoor or Trojan.

## 2.2 Vulnerabilities exploited by malware

Programs that carry out malicious behaviour need not be specially written as malware – many security breaches have involved the compromise of an existing program by an attacker. These attacks work best if the program to be compromised has a high privilege level and is accessible over network interfaces – which is why network services tend to be attacked in this way – attacks can be carried out remotely (by an attacker on the network) and physical access to the machine is not required. Malware itself often takes advantage of these vulnerabilities – for instance, most worms spread by exploiting code injection vulnerabilities in network services.

### 2.2.1 Code Injection

*Code injection* is defined as inserting new instructions into a program in memory while it is executing. There must generally be a pre-existing vulnerability in a program that makes it susceptible to various code injection techniques. However, owing to the shortcomings of common programming languages, such vulnerabilities are extremely common. In fact, the vast majority of code injection techniques exploit shortcomings in certain functions of the C programming language. Most C libraries have now been modified to implement safer versions of these functions: nevertheless, the old functions still exist and it is up to the programmer to decide which to use.

### 2.2.2 Buffer Overflow

The C programming language has no mechanism for determining whether an array access is out of range. Thus, it is easy to accidentally read or write past the end of the array. When this happens, a *buffer overflow* is said to have occurred. Reading past the end of an array will cause no problems – though what is read will be undefined – but if a program writes past the end of an array, the results can be far worse. On the Intel 80(x)86 architecture, function calls are implemented by having the processor push the address of the next instruction onto the stack, call the function, then pop the return address off the stack and jump there. Because function local variables (including arrays) are allocated from stack space, if a local variable "overflows", the return address on the stack can be overwritten. When the function finishes, the processor tries to continue execution from the address on the stack, which has been overwritten. This type of buffer overflow is therefore termed a stack overflow [AlephOne, 1996].

Generally the new value of the return address will not point to a valid memory location, so a crash occurs. If, however, the attacker can manage to overwrite the return address with the address of some executable code, she can get the processor to execute it. The reason why this is often possible is that C has no inbuilt **string** data type – to get a string, it is necessary to use an array of characters (**char**[]), with the end of the string being indicated by a null (zero) value. If the terminating null is absent, there's no way to tell how long the string is supposed to be. Furthermore, as strings arguments are passed to functions as a pointer, a string-handling function has no implicit way of telling how much memory has been allocated to a given string. Thus, functions which are designed to copy data into a string, or from one string to another (*strcpy(), strcat(), gets()*) are extremely liable to cause buffer overflows.

Once an attacker has discovered a vulnerable program (usually a network service, as these can be exploited remotely), she can design an attack to target it. The attack generally involves sending a specially-crafted string of data to the vulnerable program. The string is long enough to cause the overflow, and is set up in such a way that the overwritten return address (for a stack overflow) points back into the attack string itself, which the attacker has filled with the machine code bytes she wishes to execute (termed *shellcode* as most early examples were designed to open a command shell). Crafting shellcode is a skillful business – it must be relocatable, as short as possible, and not contain any zero bytes, which sometimes means it must be self-modifying. However, both ready-written shellcode and attack scripts are easily available, allowing attackers with the minimum of technical skill to exploit these vulnerabilities.

The stack overflow is not the only way to inject code – other methods which can be used for code injection include heap overflows [Conover, 1999] and format string vulnerabilities [Lhee and Chapin, 2003]. The attacker does not always have to inject any new code – just changing the execution path of a program, or altering the value of key variables, may be enough for their purposes.

## 2.3 Malware Naming Conventions

According to Peter Szor, the most commonly-used malware naming scheme is the one proposed by the Computer Antivirus Researchers Organisation (CARO) in 1991 ([Szor, 2005] p36-46). However, Szor also points out that naming is a highly complicated area due to the speed with which new virus variants appear. Readers interested in the technical details are referred to Szor's book and to the references he gives there.

However, a brief overview of the naming conventions is helpful in understanding the names given by virus encyclopedias or antivirus products.

Szor gives the following definition for a malware name corresponding to the CARO scheme.

```
[<malware_type>://][<platform>]/<family_name>[.<group_name>]
\[.<infective_length>][.<variant>[<devolution]][<modifiers>]
```

In the definition, 'malware_type' refers to the type (virus, Trojan, etc.). The 'platform' field is one of a list of accepted codes for target platforms, examples being 'W32' for 32-bit Windows, 'Linux' for Linux, 'OSX' for Mac OS X, and so on. The 'group_name' field allows families of similar viruses to be grouped together, and the 'infective_length' field specifies the infective length for a file-infecting virus. The 'variant' field (generally a letter) distinguishes between minor variants with the same infective length. The remaining fields are less important, and their description is omitted for brevity.

The following are the names of some programs in the author's malware collection, as well as viruses encountered by the author and not retained:

- Net-Worm://Win32.Welchia – a Windows network worm.
- Backdoor://Win32.ciadoor.121 – a Windows backdoor Trojan.
- Virus://Win32.Parite.B – a Windows memory-resident file-infecting virus.

However, the field of malware research is made vastly more complicated by the fact that the same piece of malware may be given different names by its creator and by different antivirus companies.

## 2.4 Detection and Prevention Techniques

In Chapter 1 section 1.3, four separate approaches (Detection, Identification, Prevention, and Recovery) are set out. It is possible to approach this differently – in "Foundations of Computer Security", David Salomon expresses the problem in terms of a set of four goals that an ideal anti-malware program should achieve (see [Salomon, 2006] p147)

**David Salomon's Goals for Ideal Antivirus Software**

1. To detect all known viruses and malware that currently exist in the computer, advise the user on each occurrence of rogue software discovered, and help the user to delete them.
2. To detect unknown viruses.
3. To scan incoming email, all downloaded files, and any removable storage devices inserted into the computer, and detect all known viruses and malware in them.
4. To record all activities in a log file.

Salomon then goes on to enumerate three types of anti-virus measures, which exactly parallel the ones that were described in Chapter 1, though Salomon's discussion is more general, considering user-led activities (such as taking care when opening email attachments) as well as those carried out by programs:

1. Virus-specific detection methods (Identification).
2. Generic virus detection techniques (Detection).
3. Virus preventive techniques (Prevention).

Of course, a major problem with obtaining information about the workings of extant anti-malware software is that companies are extremely loath to publish details – and not just for reasons of commercial confidentiality. As David Salomon writes,

> "....the task of anti-virus software is complex. There are many hundreds of viruses and new ones appear all the time. Current computers have huge-capacity disk drives and it is common to have hundreds of thousands of files on a single disk. Anti-virus software must therefore contain large tables with information on many viruses and has to employ clever algorithms and shortcuts in order to scan and disinfect an entire disk in a reasonable period of time. It is therefore no wonder that the makers of such software keep the details of their programs secret. The secrets are kept not just from competitors (and from this author) but also from writers of future viruses..." – [Salomon, 2006] p147–148

For this reason, the information in this section should be considered as plausible inference, although it is supported with published sources wherever these are available.

### Signature Scanning

Signature scanning is a process whereby a unique 'signature' is computed for a given virus or other malicious program. This is generally done by the manufacturers of antivirus products, who equip their software with a set of such signatures that is periodically updated. When the antivirus product scans a program it checks to see if the signature of a known malicious program is present. Of course, the disadvantage of this method is that it can only detect malware that have previously had signatures generated. However, signature scanning lends itself to detecting all types of malicious program and not just viruses, which is probably one of the reasons for its continued popularity.

Another commonly-used process based on signatures is the detection of changes to programs via checksumming. A cryptographic hash function such as MD5 is used to generate checksums for all the program files on a 'clean' system. These are stored in a secure location. Periodically the checksums for all programs are recalculated and compared with the stored values. If a virus infects a program, its checksum will change, allowing detection. However, this only works for file-infecting viruses, and assumes that the virus is unable to compromise the checksumming process or the stored values, which necessitates cumbersome measures to protect them. Checksums are therefore more commonly used as part of general integrity-checking processes (such as providing MD5 checksums for downloadable files) than specifically for detecting malware.

## 2.4.1 Heuristic Scanners

Although most commercial virus scanners rely on signatures, some also include an element of 'heuristic' scanning ability. Heuristic techniques involve being able to tell when a program is likely to be malicious based on extensive analysis of malware and 'legitimate' programs. Heuristic technology may be passive (based on scanning program files only) or active (watching what a program does when it runs), but in current commercial virus scanners it is usually passive. Peter Szor lists a number of suspicious structural characteristics an executable file might have if it has been infected by a virus. Heuristic scanners look for these characteristics when determining if a file is infected. Note that this is not a foolproof means of detection as viruses may not have all these characteristics and may be able to conceal those they do have [Szor, 2005]. Some commercial virus scanners also monitor file execution attempts to determine whether a file which tries to execute is a known virus. If it is, infection is blocked and a warning

| **Structural Anomalies in Win32 PE Executables** |
|---|
| Code execution begins in last section |
| Suspicious section characteristics, names, or flags |
| Virtual size incorrect in PE header, or incorrect sizes in other headers |
| Suspicious gaps in sections |
| Abnormal jumps to other sections |
| Entry point in PE header points somewhere other than the .text section |
| DLL imports by ordinal |
| Patched Import Address Table (IAT) |
| Multiple file headers |
| CALL to POP, or CALL to next instruction |
| Bad checksums in DLLs |

Table 2.2: Some of the structural anomalies common in infected program files given by Szor

is displayed to the user. This is probably not genuine behaviour recognition: it is more likely that running any program causes the resident portion of the virus scanner to initiate a structural scan on the program file.

### 2.4.2 Structure-Based Detection

Traditional virus scanners, whether signature-based or heuristic, generally use structure-based anomaly detection, as they rely on being able to identify anomalies in the structure of executable files (see [Szor, 2005] chapter 11). Structural anomalies cited by Szor are listed in Table 2.2 (and are further expanded on in Chapter 5).

Structural anomaly scanning is relatively easy to implement, both for real-time memory scanning and for offline file scanning (some structural anomalies may not show up in the mapped image). The disadvantage of this approach is that viruses are becoming increasingly good at hiding structural anomalies. Also, non-viral malicious programs (Trojans etc.) may not actually have any structural anomalies.

### 2.4.3 Activity Monitors

David Salomon defines an 'activity monitor' as a software module which is designed to protect low-level system routines from being compromised by viruses ([Salomon, 2006] p150). One way to do this is to maintain a clean copy of each protected function (or a checksum or secure hash of it) in ROM. Each time the function executes, it is

checked against the clean copy, allowing any modifications a virus may have made to be detected. Unfortunately, as Salomon points out, it is impractical to require the user to perform a ROM upgrade whenever the operating system is updated. A more practical, but less secure method involves examining the internal state of the processor to determine whether the operating system or some other program has executed a protected function, and blocking execution attempts by non-trusted code. Of course, a virus writer could find ways around such an activity monitor, and thus a kind of "arms race" may result. Often at least some of the activity monitor is implemented in hardware to make it as hard as possible for malware to compromise it. Some of the new functionality invented by Microsoft as part of the NGSCB (see 2.7.4) could be considered to be activity monitors, though the motive for their development may have been Digital Rights Management rather than malware prevention.

### 2.4.4 Behaviour-Based Anomaly Detection

A more promising method could be termed 'behaviour-based anomaly detection'. This involves a detection system capable of monitoring the behaviour of all processes running on a computer and raising an alert if suspicious behaviour is detected. There are many ways to define what constitutes 'suspicious behaviour' – for example, if a word-processing process writes to a document file, it is not behaving suspiciously, but if it writes to an executable file it is. Processes that attempt to locate or write to executable files when they first load, if they are not specifically designed to do this, have probably been infected by a virus, as this is the behaviour pattern of many file-infecting viruses. Any write operations on an existing executable file are suspicious unless carried out by a debugger, process patcher, or similar tool. Any process which suddenly makes system calls that it has never previously made is suspicious (this assumes that a profile of the system calls made in normal use is available). Unfortunately, monitoring this kind of behaviour requires very low-level access to the operating system.

## 2.5 Detection Countermeasures

As malware detection techniques have advanced, malware authors have found new ways to frustrate them. This section illustrates some of the more common ones.

### 2.5.1 Heuristic Frustration: Concealment

Virus authors are well aware of the suspicious characteristics looked for by heuristic virus scanners (see above) and will often design their creations so that they avoid or conceal these characteristics.

### 2.5.2 Polymorphism and Metamorphism

Most successful executable infectors are 'polymorphic' because each time the viral code is copied to a new target it is altered. This is possible because for any given operation there are generally a large number of equivalent machine instructions. For an encrypted virus, the decryption code is probably the easiest to detect, so a polymorphic virus will change its decryptor for each new infection. The earliest polymorphic viruses contained a number of different decryptors and simply used a different one each time, but more sophisticated techniques are used – random junk instructions can be inserted, equivalent instructions can be swapped and the order of execution (decryption) can be changed. Metamorphic viruses use polymorphism on the entire virus rather than just on the decryptor. Whereas polymorphic viruses generally decrypt to a constant virus body in memory, with metamorphic viruses each virus body is different.

The idea of this is to frustrate antivirus programs that work by signature scanning. The success of the virus depends on how good the altering algorithms are – for the best polymorphic and metamorphic viruses, different infections have almost no code in common, and so are very hard to detect.

### 2.5.3 Encryption and Packing

If a virus is encrypted, it is all the more difficult for a virus scanner to learn anything about it. Encryption which is trivial for a human to break (i.e. XOR) can nevertheless be very effective for frustrating virus scanning. However, with the advent of operating-system support for serious encryption, some viruses are using 'real' encryption algorithms such as IDEA. Such encryption can never be totally secure since the virus must always carry the key somewhere, but if the virus implements an odd algorithm it can be very tiresome to analyse as the analyst must re-implement the algorithm. Decryptors are often polymorphic (see above) and will decrypt portions of virus code in a seemingly random order (via a mathematical pseudorandom permuta-

tion function) to confuse heuristic decryption detectors (see Peter Szor's description of the W32/Simile virus in [Szor, 2005] p282).

### 2.5.4 Retroviruses

Some viruses are designed specifically to attack antivirus or security products running on a target computer. Such viruses are known as retroviruses (the analogy being with biological retroviruses such as HIV which attack the immune system). Additionally, other types of malware may include retrovirus-like functionality (for instance, some spyware has been known to search for popular anti-spyware utilities and attempt to deactivate them).

## 2.6 Malware and the Law

Is the authorship and dissemination of malware illegal? Evidently this depends on the legal jurisdiction. In the UK, virus authors such as The Black Baron (Christopher Pile[7]) have successfully been prosecuted under the Computer Misuse Act 1990, even though this is more concerned with humans gaining unauthorised access to computer systems than with malware. According to Buzzard,[8] Pile was charged with five violations of Section 1 of the Act (unauthorised access) and five counts of violating Section 3 (unauthorised modification). He was also charged with inciting others to spread his viruses. He pleaded guilty to all charges and was sent to prison. More recently, the Computer Misuse Act was amended by sections 35–38 of the Police and Justice Act 2006. These amendments were necessary because technological changes since the original Act meant that some malicious actions (such as denial of service attacks) were not covered. Thus it is now an offence not only to gain unauthorised access to computer material, but also to "enable any such access to be secured" which would presumably cover using malware to gain access or to steal data. DoS and DDoS attacks are outlawed by Section 36, which amends Section 3 of the original Computer Misuse Act and makes it an offence to perform an unauthorised act which is intended "to prevent or hinder access to any program or data held in any computer", even if the hindrance is only temporary. Section 37 adds a new offence to the Computer Misuse Act (creating a Section 3A) – that of making, adapting, supplying or offering to supply

[7]R v Pile, 1995

[8]"The Computer Misuse Act 1990 – Loopholes and Anomalies" by Keith Lawrence Buzzard, reprinted in [Blyth and Kovacich, 2001]

any article which is intended or believed to be for use in committing or assisting in the commission of a computer misuse offence. Furthermore, 'article' specifically includes "any program or data held in electronic form". Thus, it appears that writing or disseminating malicious programs is now illegal in the UK.[9] However, it is highly unlikely that legal redress will be possible for the average victim of a malware attack unless a specific, local perpetrator can be proven to have authored or introduced the malware.

## 2.7 Literature Review

### 2.7.1 Introduction

Malware is often by its nature self-replicating. The detection and classification of malware is therefore somewhat analogous to the tasks of detection and taxonomy of living systems in biology. Thus one might seek to classify programs by their phenotypical (or genotypical) characteristics, i.e. their structure, or by their behavioural characteristics. This thesis is not concerned with the *taxonomy* of malware (albeit that this is a very interesting topic in its own right).

There is an interesting contrast between research into malicious programs and other areas of computer security research. Whilst many conferences and journals exist for computer security-related topics, relatively few papers on malicious software are published. There are several reasons for this, a few of which will now be given. Firstly, in all security-related research (but malware research particularly) it is necessary to ensure that potential attackers do not gain valuable information from disclosure. Secondly, a great deal of anti-malware research has been carried out by private AV companies rather than by academics, and such companies often consider their results 'commercially confidential'. It is not terribly easy to find reputable sources of information about malicious programs, and it is necessary to turn to the "underground" publications of malware authors themselves, as such authors are unlikely to publish their results in a peer-reviewed journal. [10]

[9] Persons found guilty may receive up to 12 months imprisonment in England or Wales and up to six months in Scotland.

[10] A cynical possible third reason for the "cliqueyness" of the malware field is that the malware authors and anti-malware researchers fear the disruption of their comfortable symbiotic relationship....

### 2.7.2 Early Research and Theory

According to Cohen [Cohen, 1987], the term 'virus' as applied to a replicating program which spreads by infecting other programs was coined by Leonard Adleman (who is also famous as one of the three designers of the RSA encryption algorithm) at a security conference in 1983. However, it has since emerged that the term was used eleven years earlier in a novel by David Gerrold (see [Salomon, 2006] p37), though the program described by Gerrold appears, at least from the brief excerpt given by Salomon, to have been closer to a worm than a modern virus. Various people have claimed to have developed virus-like programs before Cohen, but none are known to have survived. The term 'virus' had also been used by Gunn to denote a situation where an APL program modifies its interpreter [Gunn, 1974].

It is notable that, out of all the different types of malware, early research (e.g. prior to the mid 1990s) concentrated most heavily on viruses. Factors which might have influenced this are several, but the main one is certainly that until connectivity became universal, viruses (including boot-sector viruses) were the most prevalent type of malware and caused the greatest number of problems for users of personal computers. The situation today, where pure viruses are relatively uncommon compared to worms and data-stealing Trojans, occurred coincidentally with the rise of a large base of relatively insecure but highly connected hosts (usually home PCs) and the associated growth in transactions involving valuable private data taking place over the Internet. It is also arguable that viruses are the most interesting type of malware from a research point of view; in contrast to Trojans and worms, viruses are more elegant and more challenging to create and study.

The "Trojan horse" idea was first mentioned in R.R. Linde's comprehensive paper *Operating System Penetration*[Linde, 1975]. While the idea of replicating programs was known, they were seen as an interesting academic idea rather than as a security threat. For example, Cohen cites experiments done at Xerox (described in [Shoch and Hupp, 1982]) which used primitive 'worms' for performing distributed computation – though at that time it was known that such worms could accidentally get out of control and cause denial of service. The first viruses for popular microcomputers such as the IBM PC and Amiga did not appear until about 1987. In 1988 Leonard Adleman wrote *An abstract theory of computer viruses* [Adleman, 1988 1990] in which he modelled viral infections using recursive functions instead of Cohen's Turing machines.

Cohen's 1984 paper *Computer Viruses – Theory and Experiments* [Cohen, 1987] is a landmark in early virus research. In the paper Cohen describes how replicating viruses

can be a threat to security, and neatly proves that the problem of detecting with certainty whether or not a given program is a virus is undecidable. Cohen's proof – a diagonal argument – runs as follows: in order to determine whether a given program is a virus, it is necessary to determine whether it infects other programs. Cohen assumes the existence of some program $V$ which is a virus, and a hypothetical decision procedure $D$ which, on being given some representation of a program as input, determines whether or not the input program infects other programs. $D$ returns true if the input program is a virus and false if the input program is not a virus. So running $D(V)$ would return true. Cohen then modifies the virus $V$, creating a modified version denoted $V'$. $V'$ is written in such a way that it only infects other programs if $D(V')$ returns false (in other words, if the result of calling $D$ with $V'$ as the input program is false). Assuming there is such a $D$, $V'$ is easy to write. However, $D(V')$ will only return false if $V'$ is not a virus. Therefore $V'$ will only behave as a virus if the procedure $D$ decides that it is not one. If $D$ decides that $V'$ is a virus, it will not infect other programs and hence is not a virus. This is a contradiction: therefore, no such $D$ can exist.

Cohen's paper also described practical virus experiments that were implemented on a VAX 11/250 during a computer security seminar in 1983. The virus was implanted (funnily enough) in the 'vd' utility, a graphical file display program. In each of the five attacks all system rights were granted to the attacker in under an hour. This perturbed the system administrators to the extent that shortly afterwards all further experiments were banned. In follow-up papers [Cohen, 1989] Cohen developed a model for viruses based on Turing machines. Leonard Adleman developed an alternative model based on recursive functions, which is described in his 1988 paper cited above. Recently, Adleman's definitions have been improved upon by Zhihong Zuo and colleagues at the University of Electronic Science and Technology of China. A recent paper of theirs [Zuo *et al.*, 2006], following their earlier work [Zuo and Zhou, 2004], proposes a 3-type classification scheme for viruses depending on their "imitation behaviours", where imitation is defined as "a property upon which computer viruses rely to behave like the original programs." The authors then go on to prove that the sets of Type 0 and Type 1 viruses are $\Pi_2$-complete whereas the set of Type 3 viruses is $\Pi_3$-complete.

Whilst these theoretical results are interesting, it is not clear that they have made any significant impact on practical detection methods.

### 2.7.3 Practical Detection Systems

#### IBM's 'Immune System'

The first company to research ways to automate viral signature generation was IBM. Staff at their Thomas J Watson Research Center published a series of papers between 1993 and 1997 which outlined their vision of a 'computer immune system'. In *Computers and Epidemiology* [Kephart *et al.*, 1993] Jeffrey Kephart, David M Chess and Steve R White address the question of whether parallels may be drawn between biological virus epidemics and computer virus epidemics. The analogy between the biological immune system and computer security systems has also been pursued academically by a team led by Professor Stephanie Forrest at the University of New Mexico. In fact, academic interest in "artificial immune systems" is such that an international conference was set up on the subject in 2002. In 1999, however, IBM announced that it had developed *"the first commercial-grade immune system that can find, analyse and cure previously unknown viruses faster than the viruses themselves can spread"* [White *et al.*, 1999]. The IBM "computer immune system" uses signature techniques, but makes signature generation automatic. It consists of an automated method for forwarding suspected viruses via a network to a central analysis centre, where they are automatically analysed. This analysis is accomplished by allowing the suspected viruses to infect 'goat' programs, which are special programs whose characteristics are known exactly. By comparing the infected goat programs to the originals, the viral characteristics – and hence a signature – can be found. The analysis system then generates a 'vaccine' which is sent out over the network to all the protected systems. It is not clear how this would work for malware types other than file-infecting viruses, however, and the IBM strategy of deliberate infection is generally only practical in a special secure environment (such as IBM uses in its automatic analysis centre). It also seems as if this approach would be vulnerable to the countermeasures already used by some virus writers, who take pains to detect the presence of obvious 'goat' programs or emulated environments in order to frustrate antivirus researchers – and human researchers would be far less easy to fool than an automated system. This is perhaps why IBM do not seem to have proceeded much further with their system.

#### Alternative Strategies

Alternative strategies for analysing malware have been developed, using neural networks [Tesauro *et al.*, 1996], data mining [Schultz *et al.*, 2001], probabilistic detection

[Forrest *et al.*, 1994] and system call sequencing [Forrest *et al.*, 1996]. Lee, Kim and Hong [Lee *et al.*, 2004] propose a hybrid approach: firstly, all new or changed programs are divided into 'self' or 'nonself' by means of a signature process. Then, behavioural (heuristic) analysis is applied to the 'nonself' programs to determine which are viruses. Another paper by some of the same authors describes the use of genetic algorithms to improve a randomly-generated "detector set" [Kim *et al.*, 2004]. The detector set is an idea taken directly from the biological immune system. In the same way that antibodies identify 'nonself' cells by binding to certain protein sequences they contain, a computer immune system may identify 'nonself' programs by identifying bit-strings they contain. The detector set is the set of such bit-strings. It is highly probable that some of these ideas are used in modern antivirus programs, but owing to commercial confidentiality it is impossible to find out.

Some of the methods proposed in the literature will now be examined in more detail.

### 2.7.4 Static and Semantic Analysis

A common method by which human researchers analyse malicious program code is known as static analysis, presumably because the malicious code is not executing while it is being analysed. Many forms of static analysis are possible, and many tools have been developed (see [Christodorescu and Jha, 2003] and [Sabin, 2004]). Recently researchers have been using static analysis in conjunction with taxonomic techniques to determine 'family' relationships between malicious programs (e.g. [Flake, 2006] and [Lemos, 2006]). For such purposes detailed code analysis is an essential requirement, since the goal is to determine whether a given malicious program has code or structure in common with existing known cases.

A slightly different approach is to perform *semantic analysis* on malicious code [Shin and Spears, 2006]. This starts by disassembling the malicious example using a commercial disassembler. The assembly code is then converted to a graph that is a hybrid of control flow and data dependence graphs. The graph is then split into subgraphs for each program subgoal, and further each subgraph is converted into a FSM representation. By performing *inductive inference* on output strings from the FSMs (which represent execution paths) they are able to develop a 'semantic signature' for all possible malicious code in a particular class.

### Microsoft's Behaviour-Based Recognition

Very recently Microsoft have been looking at behaviour-based recognition. Their 2006 EICAR conference paper on behavioural classification [Lee and Mody, 2006] lists previous approaches to behavioural classification, pointing out that mere system or API call sequencing is often inadequate. The novelty of this approach is that, as far as is known, most current virus scanners and other existing protection methods rely entirely on *structural* anomalies in programs rather than behavioural ones. In other words, existing methods ask "Does it look malicious?" whereas this method asks "Does it *behave* maliciously?". The Microsoft researchers initially set out to automate static analysis techniques, but came to the conclusion that these were insufficient to deal with rapidly-evolving malware and so started looking at behavioural analysis as an adjunct to static analysis. They developed a method of generating vectors of behavioural data for programs, then applied a distance measure to these strings to allow classification. Some other researchers have cast doubt on the usefulness of such work, preferring improved methods of static analysis.

### Other Methods

Other research has concentrated on mathematical modelling of virus propagation, using techniques developed by epidemiologists for biological viruses. In [Chang and Young, 2005], Chang and Young use differential equation-based models to determine how the spread of viruses is affected by the nature of the Internet and how the latter might be changed to preferentially discourage the spread of viruses. Leaving the realm of viruses, Crosbie and Spafford have tried to apply intelligent agent technology to intrusion detection systems – and their papers [Crosbie and Spafford, 1995b] and [Crosbie and Spafford, 1995a] show a high degree of foresight, because at that time intelligent agent technology was in its infancy. Andrew Watkins' *An Immunological Approach to Intrusion Detection* [Watkins, 2000] brings together the agent-based approach of Crosbie and Spafford and the "self-nonself discrimination" immune-system ideas of Stephanie Forrest and others. Watkins covers only network intrusion detection (e.g. detecting hackers)– the goal of this present work is to show that these techniques can also be applied to viruses, spyware and other Malicious Programs. Spyware, indeed, is such a recent development that few if any papers cover the subject, and of the ones that do, most only mention it in passing. Most information on Spyware comes from non-academic sources (see [Webb, 2005], [Gator, 2005]).

### Trusted Computing

Finally, Microsoft and others have promoted what has been respectively known as Trusted Computing Platform Architecture (TCPA), Trusted/Trustworthy Computing (TC), or the Next Generation Secure Computing Base (NGSCB) as a solution to all computer security problems, including viruses. The NGSCB was developed by a consortium of hardware and software vendors, including major manufacturers of processor chips such as Intel and AMD. In its simplest form, as set out in [Safford, 2002], TC hardware needs to perform three basic types of function:

1. Public key functions.
2. Trusted boot functions.
3. Initialisation and management functions.

In other words, a TC chip will contain facilities for doing public-key cryptography (generating key pairs, encryption, decryption and digital signing and verification), facilities for verifying the boot process and 'sealing' data so that it cannot be accessed if the boot process changes (which can be used to protect data in the event that, for example, a hard drive is stolen), and facilities for managing the operation of the chip. All of these features will vary between different manufacturers of TC chips. Some manufacturers, or certification authorities, would retain copies of a special public key (called the 'endorsement key' by IBM) associated with each chip. This would bring certain advantages (such as allowing the system to function like a smart card and do secure e-commerce) as well as certain disadvantages (privacy concerns). Microsoft's proposed implementation of TC in Windows, originally named Palladium but also referred to as the Next Generation Secure Computing Base (NGSCB), is considerably more involved, comprising what is effectively a hardware-based authentication system which would control most or all aspects of computer operation based on permissions obtained from what might be termed 'higher authorities' (software vendors and content owners). Furthermore, Microsoft have proposed that all peripherals (memory, monitors, keyboards, sound cards etc.) contain encryption facilities to allow totally secure communication between themselves and the main processor. Additional motives for developing such technology may have been to make unauthorised software and content copying impossible via "Digital Rights Management" (DRM) and at the same time make it easier for Microsoft and other large companies to utilise "vendor lock-in" against users who might otherwise want to switch operating systems or application

software (see [Stallman, 2002]). This has led to the technology being redubbed 'Treacherous Computing' by its detractors. In any case, there is much reason to be sceptical of Palladium or TC as a general solution to security problems, as Anderson shows when he points out the contradiction between Microsoft's one claim that NGSCB will prevent viruses and their other claim that existing programs will run on NGSCB-enabled systems without problems [Anderson, 2003]. Microsoft initially intended to implement NGSCB in Windows Vista, but eventually decided not to do so. However, many of the DRM-enabling features (such as secure channels between hardware devices) have been implemented by Vista in any case (see below).

In 2002 David Safford of IBM Research wrote [Safford, 2002], intended as a rebuttal to the claims of Anderson and other critics. In this, Safford argues that Trusted Computing itself is separate and distinct from DRM and from Microsoft's suggested implementation of TC (Palladium), and that many commentators on the subject have confused all three technologies. Safford also points out the positive aspects of IBM's TC chip: for instance, it makes it much harder for malicious software to steal your encryption keys. Certainly, IBM's implementation as described by Safford in 2002 appears far less sinister than Microsoft's version. However, developments since 2002 have shown that where TC chips have been deployed, the motive is usually DRM (for example, Apple's new Intel Macintosh).

Though trusted computing technology is now extensively used, particularly in games consoles, as of 2006 no companies who produce or use it are representing it as anything other than advanced copy protection, or claiming that it can prevent malware. As the hackers who managed to run the DRM-protected Intel version of Mac OS X on a standard PC (and, later, to run Windows on the Intel Macintosh) have shown, hardware-based copy protection is far from foolproof, and it has the further disadvantage that protected 'content' is not usable on older hardware without the protection chip, which could be the reason it has not as yet caught on in desktop PCs.

### Windows Vista

Prior to the launch of Windows Vista in January 2007, Microsoft had promised that it would be the most secure version of Windows yet seen. The Microsoft web site claims that

> "Windows Vista is also armed with enhanced protections designed specifically for Internet use. Dynamic security protection in Windows Internet Explorer 7 helps guard your computer and your privacy against threats like malware, fraudulent websites, and online phishing scams, while new Windows Defender technology works to minimise pop-ups, slow performance, and security threats posed by spyware and other invasive programs...And, in the event that your computer is lost or stolen, a feature available in Windows Vista Ultimate called Windows BitLocker Drive Encryption will help keep your data confidential through full-volume encryption and boot-integrity monitoring."– [Microsoft, 2007]

Detractors, however, point out that Microsoft's priorities are skewed – the majority of the new security features are not for protecting the user or their data, but for protecting so-called 'premium content' – in other words, DRM.

> "The entire operating system now seems to have turned against the user. Zero tolerance drivers and regulation code will lock the system down if any type of deviance is detected. So called 'tilt bits' will signal an attack on the system if anything is found out of the ordinary. These changes won't enhance user security unfortunately as they were designed to protect only premium content. Medical data, credit card numbers, and other private things that do deserve this level of protection are completly (sic) ignored."– [Day, 2006]

Peter Gutmann of the University of Auckland has written a comprehensive technical critique of Vista's content protection system. In it, he not only shows how the many content protection features will increase hardware costs, detract from the user experience, and prevent even legitimate activities, but may even pose a security risk if attackers can exploit them for their own purposes:

> "Another unforeseen consequence of the potential for a downgrade disguised as an upgrade (that is, a driver being revoked by Windows Update) is that the whole process of updating your machine is supposed to provide benefits to the user in the form of enhanced functionality or, more pragmatically, bugfixes and security patches. Since malware attacks are invisible but a loss of playback capability isn't, if the only visible effect of an update is to reduce system functionality it incentivises users to disable updates in order to avoid this issue. The unfortunate hidden side-effect of this is that in the interests of protecting themselves from having their content-playback capabilities turned off, they're now vulnerable to all manner of malware, viruses, spyware, and so on.....Content protection features like 'tilt bits' also have worrying denial-of-service (DoS) implications..With the number of easily-accessible grenade pins that Vista's content protection provides, any piece of malware that decides to pull a few of them will cause considerable damage." – [Gutmann, 2007]

Microsoft have attempted to rebut Gutmann's critique, but have had difficulty, since the critique is extremely well-referenced – in fact, most of the information is taken directly from Microsoft's own specification documents and from those of hardware manufacturers such as ATI. At any rate, it is far from clear that any of the new security technology will prevent the spread of malware – and since Microsoft has now greatly restricted the degree to which third-party manufacturers of anti-malware tools can gain access to the internals of the operating system [Evers, 2006], users will have to rely on Microsoft's inbuilt tools.

## 2.8 Tools and Sources of Information

This section briefly lists some of the additional sources of information and tools used during this PhD but which were not covered in the review above owing to their general nature.

All programs developed during this PhD were written in C and C++ and compiled using the MinGW implementation of the GNU Compiler Collection (GCC). The development platform was Windows 2000, whereas the isolated malware test machine ran Windows XP. For information on C and C++ the two classic references [Kernighan and Ritchie, 1988] and [Stroustrup, 1997] were used. Win32 API references included Charles Petzold's "Programming Windows" [Petzold, 1999], Johnson M. Hart's "Windows System Programming" [Hart, 2005], and Rector & Newcomer's "Win32 Programming" [Rector and Newcomer, 1997]. Mark Russinovich's "Windows Internals" is a useful and comprehensive overview of how Windows works, but, as it was not specifically written as a programmer's reference, it provided disappointingly few practical details as to how programs could access system information. Donald Knuth's "The Art of Computer Programming" [Knuth, 1997] served as a useful standard algorithm reference on the few occasions one was needed.

Chapter 3 is a case study illustrating how malware could be used to assist in online banking fraud.

Chapter 3

# The law of unintended consequences: a case study

## 3.1 Introduction

The work described in this chapter is joint work with S. P. Goring and Antonia J. Jones [Goring *et al.*, 2007].

The potential consequences of successful malware penetrations have amplified as broadband technology has become ubiquitous, and progressively more commercial activities, such as shopping and banking, have become Internet based. This chapter provides an interesting illustration of how a weakness in a login procedure for an Online Banking system, combined with large scale application of keylogging malware, could have led to quite disproportionate consequences.

Traditional authentication systems used to protect access to online services (such as passwords) are vulnerable to compromise via the introduction of a keystroke logger to the service user's computer. This has become a particular problem now that many malicious programs have keystroke logging capabilities. When banks first introduced online banking services they realised this, and added features to protect users against keystroke logging. In this chapter it is shown, using a real online banking system as an example, that if these features are incorrectly implemented they can allow an attacker to bypass them completely and gain access to a user's bank account within a small number of attempts. The vulnerability was initially noticed in a particular Online Banking Service, but any system implemented in the way described is equally vulnerable.

## 3.2 Disclaimer

No illegal access took place during this research. It is generally assumed that to be in a position to *prove* that a gatekeeper system has a weakness one must have broken the law.[1] However, the work of this chapter demonstrates that this is not the case. In this case it is shown that by perfectly proper use of a system, which has implemented login in a particular way, and by *intelligent observation* one can logically *prove* a weakness without even passing the gatekeeper or entering the system [Johnson and Cobain, 2006]. Whilst in this case this can be done because of a rather trivial and presumably easy-to-fix design flaw, it seems that an interesting point of principle has been established. It should be noted that the bank in question deny that these observations constitute a security risk to their Online Banking service (see Section 3.4).

## 3.3 Unintended consequences

Karl Popper once claimed that "the main task of the theoretical human sciences ... consists in identifying the non-intentional social repercussions of intentional human actions" ([Boudon, 1977], p1). Companies or organizations such as banks which offer internet-based access to services have had to develop a number of techniques to prevent compromise of user authentication codes, which are substantially more sensitive than the average website access password. One popular technique which helps prevent access codes being recorded by keystroke loggers is to ask the user for a number of randomly-chosen characters (usually 3) from the authentication code(s). Whilst someone 'watching', either physically or via a hardware or software-based keystroke logger, would obtain the account ID (and extra information such as the user's birthday), only a few characters from the authentication code would be harvested and their positions not be known. A keystroke logger would still eventually capture all pieces of the code, but without the positional information the attacker has no data to enable these pieces to be put together within the usual three login attempts. So far so good. However, sometimes, what may seem to be a useful additional security measure can have unforseen consequences. Here it is shown how an apparently minor detail of implementation can, if flawed, effectively entirely negate the anti-keylogging measures; thus providing an interesting example of the 'law of unintended consequences'.

[1] This has the effect of discouraging people who discover such flaws from reporting them because they fear (often correctly) legal reprisals. Popper might have called the topic of this chapter "an unintended consequence in a closed society".

The vulnerability herein described allows an attacker to know beforehand which character positions the victim will be asked for on his next login attempt. When the victim next logs in, the attacker can view the characters which correspond to these positions with the keystroke logger, and repeat the process to get more characters from the authentication code. In general at each such move (see section 3.7) *either* the attacker gains entry *or* she learns at least one new digit (and its position). So it's a win-win situation for the attacker in terms of information gain. In this particular case the maximum authentication code length is 10 digits. After the first move three digits (and their positions) are known, this inevitably means that in at most *eight* more moves the attacker is *guaranteed* to gain access. However, probabilistically the attacker will gain entry very much faster than this, typically within three to five moves. Keystroke loggers are not hard to obtain. Hardware loggers which are designed to look like various kinds of keyboard adapters may be purchased online relatively cheaply. These require physical access to the computer at least once, but they can be used on any computer, need no software, and are almost undetectable (other than by physical examination). Some of the more advanced types allow their data to be accessed remotely via a radio link and so require only a *single* physical penetration.

Software loggers, on the other hand, are available free (or can be written by a relatively competent programmer), in principle can be remotely queried over the Internet, and in fact form a component of many already-existing malicious programs.

## 3.4 The login procedure

The Online Banking service previously mentioned will be used as an example. This system requires a customer to select an authentication code which must consist of between six and ten numeric characters (0-9). On logging in to the service the customer must enter a banking ID number and their date of birth. These must be entered in full every time the user logs on. The customer is then prompted to enter three characters whose positions are chosen randomly from this code. It should be noted that the bank in question does not accept that these observations imply a security risk, see [Guardian, 2006].

## 3.5 The Vulnerability

Assume the user makes a mistake in entering the three authentication code characters. When this happens, he is presented with a failure message and must re-start the login process. However, at this point the system designer has a choice: the system can either ask for the *same* set of three digits, or ask for a *new* randomly generated set.[2]

- The situation analyzed here is when, under these circumstances, the user is asked for *exactly the same characters* he was asked for previously - in other words, if he is asked for the first, fifth and sixth characters and gets it wrong, he will be asked for the first, fifth and sixth characters when he tries again.

For such an implementation the requested character positions only change after a successful login, and this remains the case even if the next attempt is made from a different computer at a different time.

It is thus possible for an attacker, by making a login attempt, to gain advance knowledge of which digits the user will be asked for the next time they log in. This allows the attacker to reconstruct the authentication code from the keystroke logger transcripts. The ID and date of birth will be captured by the keystroke logger the first time the victim logs in. The attacker must know these to proceed, and they also provide a convenient search string to allow her to locate the pertinent part of the logger's captured data.

Obviously, if the logger is present on the victim's machine for long enough it is certain to capture all the digits in the authentication code. However, the attacker has no access to positional data – she will have sets of 3 digits, but no idea of their sequence in the authentication code, or even of their uniqueness (the same digit may occupy more than one position in the code). Even if an attack is feasible based on this idea, it is likely to be vastly slower and less certain of success than the attack described here.

It has been argued that this stratagem, of not requesting a different set of characters on each retry, acts as a defense against phishing sites (in this case a website that masquerades as the legitimate site and attempts to garner users login details, i.e. website forgery). It appears to the authors that this argument is flawed. It is only a defense if the *user* is aware of the fact that if they mistype their authentication code the genuine site *will not* request a different set of characters. The authors submit that

[2]Or, indeed, do something else entirely different.

many users would not draw such a distinction in practice and act on it, even if they had once been told of it.

There are other technical measures against phishing websites (see [Wikipedia, 2007]). Moreover, granted sensible precautions (e.g. such as registering similar domain names so they cannot be used for phishing), in the final analysis, assuming customers have been warned, it does not seem (to the authors) reasonable to hold a company responsible because a customer has entered their authentication details on some other website. In any event this seems to be an increasing problem. Reports suggest that in the United Kingdom losses from web banking fraud, mostly from phishing, almost doubled to £23.2m in 2005, from £12.2m in 2004 [Finextra, 2006], while 1 in 20 users claimed to have lost out to phishing in 2005 [Richardson, 2005]. So phishing is undoubtedly a serious issue.

## 3.6 Attack Methodology

The attacker must be able to introduce a keystroke logging device to the victim's computer. For hardware loggers, an initial physical access is required - though software loggers could be injected using a virus or various other non-physical methods. Methods based on 'social engineering' are often very successful – see [Stasiukonis, 2006] for a particularly novel study in which several USB pen drives containing automatic-installing keylogging software were left lying around a company car park, and proceeded to infect the entire company when curious employees picked them up and plugged them into their computers to see what they contained.

The attacker first waits until the keystroke logger has captured the ID number and date of birth (plus three authentication code digits whose position is not known which, although they do provide useful information, will be ignored in this context). Then she makes a login attempt, noting the positions of the three requested digits. This attempt will in all probability fail (ignoring the extra 'negative' information gained here), but the attacker now *knows* that the victim will be entering his next three digits at those specific positions next time he logs on.

When the attacker sees from the logger transcript that the victim has logged on for the second time, she notes down the digits, pairing them with the position data she already has. She now makes a second login attempt, again noting the digit-positions requested. This cycle is repeated, either until the attacker is asked for a set of digit-positions she already has (and can thus access the account), or until the attacker has

the entire authentication code. Of course, if the code is a recognizable number, such as a date, then the attacker may be able to deduce it without needing to recover all the digits.

The sequence of events is then as illustrated in the flow chart of Figure 3.1. Note that the process does not have to 'END' at the stage indicated in the flow chart (the first successful login), it could continue until (with high probability) the *complete* authentication code had been retrieved.

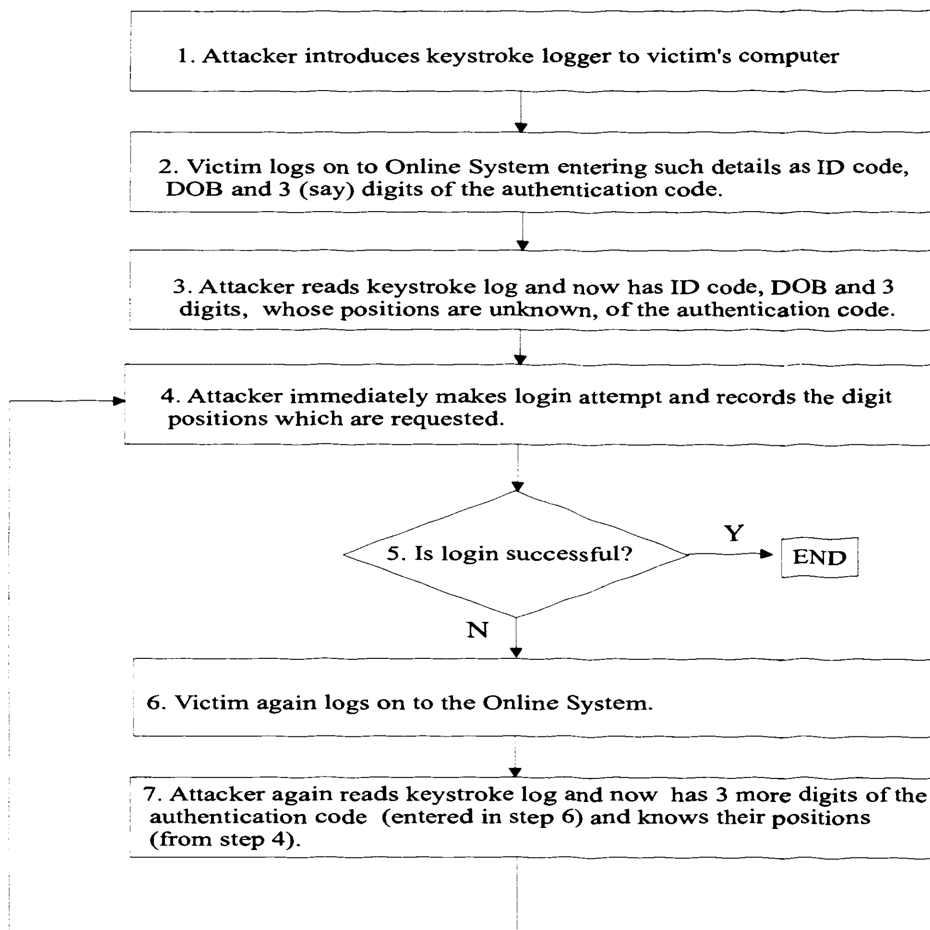



Figure 3.1: The sequence of attack steps described as a flow chart.

The time frame within which login can be accomplished, or the complete code recovered, depends on how frequently the victim uses the service and the dedication of the attacker.

It has been argued that this is a complex attack which would be time consuming and dependent on frequent user logins. Whilst perhaps true for a single target the authors feel that this objection misses the essential point. The methodology is easily automated by the dissemination of key-logging Trojans. As these payloads are successfully inserted

and report back via the Internet to their designer, a large database of victims could be automatically constructed and eventually exploited (perhaps simultaneously), all without any further work on the part of the attacker. It becomes a matter of statistics: some login credentials will be harvested and some will not, but with patience on the part of the attacker the scale of the eventual damage could easily be significant.

In case this seems far-fetched, it must be noted that at least one exploit with approximately similar features has already taken place, see [Richardson, 2005] and [Leyden, 2005]. In the United States in 2005, a Miami businessman sued his bank after $90,000 was lifted from his firm's online banking account following a computer virus attack [Ley, 2005] in which it appears authentication details were compromised by a Trojan (CorelFlood).

## 3.7 Probabilistic Analysis

In the following, a 'move' consists of the following sub-steps:

1. Victim logs in to the Online Banking system.
2. Attacker notes new digits.
3. Attacker makes login attempt and notes which positions are requested.

| **Number Known** | **Overlap** | | | |
|---|---|---|---|---|
| k | $d = 0$ | $d = 1$ | $d = 2$ | $d = 3$ |
| $k = 3$ | 0.2917 | 0.1525 | 0.175 | 0.0083 |
| $k = 4$ | 0.1667 | 0.5 | 0.3 | 0.0333 |
| $k = 5$ | 0.0833 | 0.4167 | 0.4167 | 0.0833 |
| $k = 6$ | 0.0333 | 0.3 | 0.5 | 0.1667 |
| $k = 7$ | 0.0083 | 0.175 | 0.525 | 0.2917 |
| $k = 8$ | 0 | 0.0667 | 0.4667 | 0.4667 |
| $k = 9$ | 0 | 0 | 0.3 | 0.7 |

Table 3.1: Probabilities $P(10, k, d)$ for the overlap

Suppose the authentication code length is $n$ ($6 \leq n \leq 10$) and that the values (and positions) of $k$ digits are known. $P(n, k, d)$ denotes the probability that, at the next

move, exactly $d$ of the three positions requested are amongst the $k$ already known. $d$ is termed the *overlap*. So $P(n, k, d)$ is the probability that at the next move the overlap will be $d$. $P(n, k, d)$ may be calculated as follows:

$$P(n, k, d) = \frac{\binom{k}{d}\binom{n-k}{3-d}}{\binom{n}{3}} \tag{3.1}$$

This is because of the total $\binom{n}{3}$ ways of choosing 3 positions from $n$ precisely $\binom{k}{d}\binom{n-k}{3-d}$ have an overlap of $d$ with the $k$ known positions. Calculating $P$ for different values of $d$ and $k$ produces Table 3.1. Using this table the relevant probability tree can be constructed for the worst case in which $n = 10$. The first few nodes are shown in Figure 3.2.

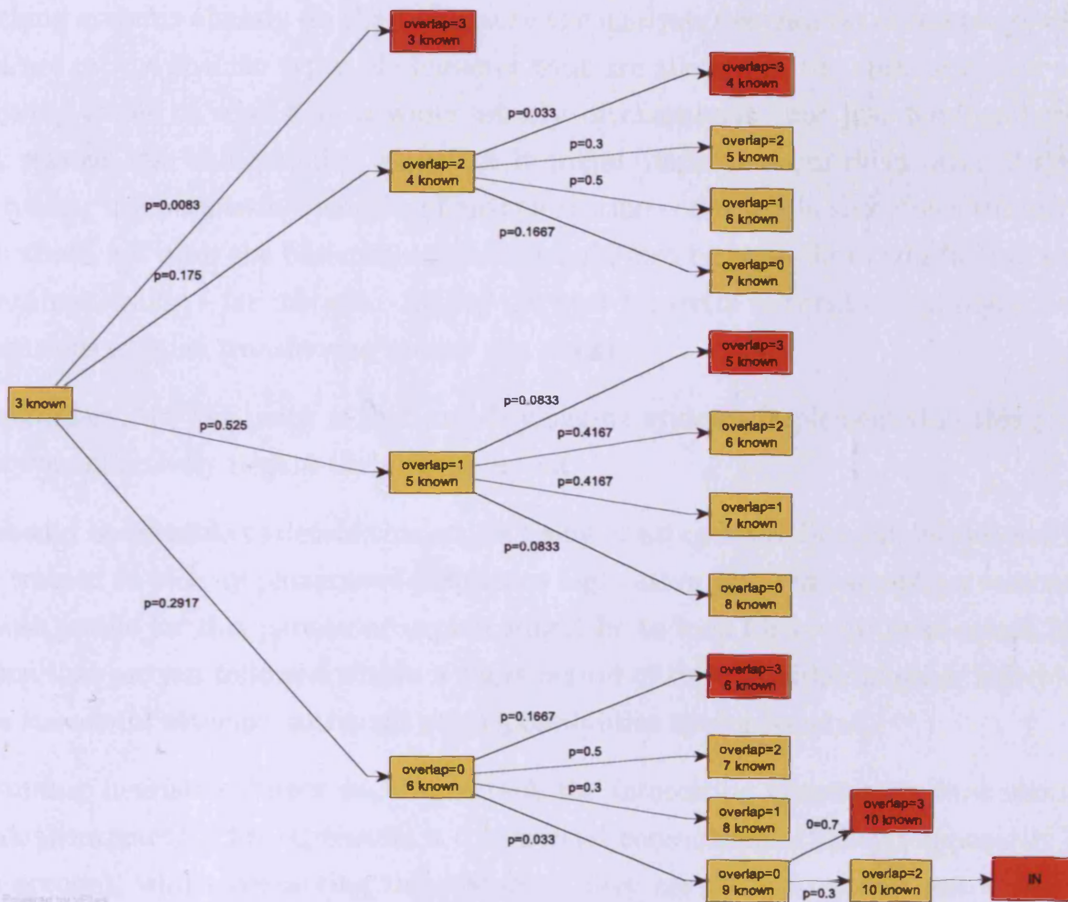



Figure 3.2: The first few nodes of the probability tree. Terminal nodes at which access has been gained are shaded (red). Only a few nodes beyond level three are shown.

Complete analysis shows that the full tree has maximum depth eight steps from the root node. Hence an attacker is *guaranteed* access within nine moves. However, the

probability of tracing this path through the tree is 0.0000879, and so needing eight steps is exceedingly unlikely. The *most probable* exact number of moves is five, with a probability of just over 0.5. This is the worst case, since it is assumed that the user has chosen an $n = 10$ digit random authentication code. If the user has chosen a shorter authentication code, or a code whose structure may be predicted (such as an 8 or 6-digit date), then access could be gained in even fewer moves.

## 3.8 Conclusions

The vulnerability described here may easily be removed by ensuring that the system always asks for a new set of characters whether or not login is successful (many online banking systems already do this). Because the analysis depends on character *positions* and not on the specific types of character that are allowed in the authentication code, allowing codes to consist of a wider variety of characters (not just numbers) would not remove the vulnerability, although it might improve security in other respects. Increasing the permissible lengths of authentication codes would slow down this attack, but would not alter the basic situation. It would also be possible to add further stages of authentication - for instance, asking the user for extra authentication digits before operations such as transferring money can occur.

In summary, the key point is that anti-keylogging systems implemented in this particular way effectively negate their entire intent.

It should be possible to detect this attack using existing heuristic analysis systems that are trained to pick up patterns of suspicious login attempts. For example, a reasonably robust profile for this particular exploit might be to look for two or more single failed logins that are not followed within a short period of time by either another failed login or a successful attempt, although other possibilities spring to mind.

Assuming heuristics detect such an attack the interesting question is: how should a bank then react? This represents a commercial conundrum. If they temporarily lock the account, whilst contacting the customer, they are going to annoy customers, and so perhaps lose money. If they don't react, but could have done, then they'll take the liability.

Banks are not in the business of producing perfect security, what they want is security that is as low cost as possible yet still adequate. Like webhosts in regard to child pornography, the current state of the law might thus tempt banks, who seek to do the right thing by their shareholders, to adopt a '*don't look - so we don't know*' approach.

More generally, this is one example of the flaws emerging in single-factor authentication using end-points of unknown security. For others, one might look at the E-gold Trojan, see [LURHQ, 2004] and [Baseline, 2006]. This Trojan automates the burden of siphoning money from the accounts and does it from the victim's own computer. Because it uses the victim's established SSL session and does not connect out on its own, it can bypass personal and corporate firewalls and evade IDS/IPS devices:

> In other words, the new Trojan programs do not have to trick victims out of revealing their password. Instead, they wait for the victim to perform his normal banking business. While the victim checks his bank balance, the Trojan silently siphons money out of the account. –[Counterpane and Messagelabs, 2005]

Other, even more creative, approaches are the Blue Pill[3] [Rutkowska, 2006] or Metasploit an Open Source project inspired by H. D. Moore of *BreakingPoint Systems*:

> Meterpreter, short for The Meta-Interpreter is an advanced payload that is included in the Metasploit Framework. Its purpose is to provide complex and advanced features that would otherwise be tedious to implement purely in assembly. The way that it accomplishes this is by allowing developers to write their own extensions in the form of shared object (DLL) files that can be uploaded and injected into a running process on a target computer after exploitation has occurred. Meterpreter and all of the extensions that it loads are executed entirely from memory and never touch the disk, thus allowing them to execute under the radar of standard Anti-Virus detection. –[Scape, 2004]

The argument has come full circle. In contrast to a 'closed worlds' philosophy, that instinctively hides weakness, Moore exemplifies the 'open worlds' approach (which arguably is now inevitable) of the Internet driven information space now inhabited: vulnerabilities published are vulnerabilities more likely to be addressed.

Having no real assurances regarding the security of the user-point machine poses real problems for secure transactions which rely on purely software solutions. It is also the case that the time interval between when a vulnerability is published and when exploit code appears has shrunk, as criminals try to best take advantage of the 'window of exposure' before systems are patched or signature based Virus-checkers are updated, so that even knowledgeable and conscientious users cannot be confident regarding their system integrity.

[3]Presumably a reference to the blue pill which Morpheus offers Neo in "The Matrix".

This observation in part motivates the work of this thesis. How can computers or networks be defended against penetration and exploitation by hitherto *unseen* instances of malware? Perhaps by using machine learning to determine those structural or behavioural generic characteristics which might be used to *identify* malware, even though the system has not previously seen this particular instance? The next chapter will examine some machine learning techniques which might be applicable to this problem.

In any event these examples, together with the particular login issue discussed in this chapter, raise doubts about the wisdom of having *any* sensitive system online with purely software access, and perhaps about online banking in general. One cannot always foresee the consequences of new technical developments. As bandwidth increases, and hardware memory chips becomes smaller and incredibly capacious, today's keystroke loggers might easily, for example, become tomorrow's 'keystroke plus screen' loggers.

It is, of course, possible that technology has outstripped the need for malware to exploit the vulnerability described here. The E-Gold Trojan referenced above, which siphons money out of a user's account by hijacking an existing session, is both easier to write, and would possibly yield a greater return, than an attack based on the above method. The main value of the above method is that it can be targeted with great precision (this was used as an argument against it, but in fact it could be construed as an advantage).

The exploit described in this chapter is merely one example of the damage that can be done by malware. The next chapter begins the investigation of automating the recognition process for malware by giving an overview of the classification methods and machine learning systems used in this work.

# Chapter 4

# Machine learning classification methods and systems

## 4.1 Introduction

This chapter will discuss various classification methods used in machine learning, specific implementations in learning systems, the eventual methods and implementation chosen, and the reasons for this choice.

In [Tan *et al.*, 2006], four core tasks in data mining/machine learning are identified: cluster analysis, predictive modelling, association analysis and anomaly detection. It was decided early on to use a methodology based on *predictive modelling* (in particular classification) as opposed to one based on anomaly detection. This is because the task is not simply to identify that a given program/process differs from others, but to identify any such differences as specific to a given class of malicious software. It is possible that an approach based on anomaly detection might be useful as a preliminary screening process, but it is not necessarily true that, assessed against an arbitrary scale, the set of 'anomalous' programs/processes exactly corresponds with the set of malicious processes.

## 4.2 Classification methods

Tan defines a classifier (or classification method) as "a systematic approach to building classification models from an input data set." ([Tan *et al.*, 2006] p148 – "General

Approach to Solving a Classification Problem") This section will discuss three classification methods commonly used in machine learning, and explain why only two of them were eventually chosen. The reason for limiting the discussion to these three is that, given the number of classification methods and variants in existence (for instance WEKA Explorer lists 70 different classifiers), it was necessary to decide at the start that certain classifiers would be used to avoid spending too much time evaluating the merits of different classifiers.[1]

### 4.2.1 Decision Trees

Decision theory defines a decision tree as "...a graph of decisions and their possible consequences (including resource costs and risks) used to create a plan to reach a goal". Automated decision trees are used in machine learning systems. Here, the tree represents a mapping of observations about an item to conclusions about the item's target value. The process of inducing a decision tree from data are called 'decision tree learning'. Decision trees are divided into two types: *classification trees* and *regression trees.* Classification trees are used where the decision output is binary (either/or), whereas regression trees can provide a real number output.

Computer-based decision tree learning systems can be described as a mechanised version of '20 Questions' or similar children's game. Players must discover the name of some object by asking a series of questions. The idea is to minimise the number of questions. Therefore questions should be selected carefully, so as to maximise the information gain at each step. One can visualise the process as a tree in which a question is asked at each node. The root node represents the universe of all possible answers and each question splits the answer set into subsets – each subset represented by a child node.

Having decided on a 'splitting criterion' at each node of the tree most implementations of such systems are 'trained' with a list of known cases and the associated 'answers' for each. Once the trees are built, they can be used to classify unknown cases. The standard method, as described in [Quinlan, 1989], and in somewhat more detail in [Quinlan, 1993a], of constructing a decision tree classifier from training data are as follows. It is assumed that the cases in the training data set are of known classes with fixed attributes.

[1] Even [Tan *et al.*, 2006], a textbook devoted to data mining, requires just under 200 pages to cover only six types of classifier: decision trees, rule-based classifiers, nearest-neighbour classifiers, Bayesian classifiers, Artificial Neural Networks, and Support Vector Machines.

- If all training cases belong to a single class, the tree is a leaf labelled with that class.
- Otherwise,
  - select a test (i.e. a splitting criterion), based on one attribute, with mutually exclusive outcomes
  - divide the training set into subsets, each corresponding to one outcome
  - apply the same procedure to each subset

Whilst it is true that any test will eventually result in a complete partition of the data into single-class subsets, not all such trees will reveal information about the domain structure. The ideal tree has many cases at each leaf, or as few partitions as possible – so the ideal would be to choose tests at each size such that the final tree is as small as possible. Since there is generally a very large number of possible trees, exhaustive search for the best one is not a viable solution. Indeed it is well known that the problem of finding the smallest decision tree consistent with a given training set is NP-complete [Hyafil and Rivest, 1976]. It is therefore necessary to employ some criterion for splitting each node when constructing the tree. The criterion commonly used is to split so as to maximise 'information gain' (see Quinlan's books and papers for more information on the criteria used by C4.5). WEKA implements several different decision-tree algorithms, including some that are functionally identical to ID3 and C4.5.

### 4.2.2 Naive Bayes

Bayesian classifiers are probabilistic classifiers based on Bayes' Theorem [Bayes, 1763]. There are two main types: Naive Bayesian Classifiers and Bayesian Belief Networks. For this work only simple Bayesian classifiers will be used. Following is a brief description of the use of Bayes' theorem in classification. The treatment in ([Tan *et al.*, 2006] pages 229–240) will be followed.

Assume that the attribute set is denoted by $X$ and the class (descriptor) variable by $Y$. If the evidence $X$ and the descriptor $Y$ are treated as random variables, it is then possible to denote the *prior probability* of $Y$ as $P(Y)$, and the *posterior probability* of a $Y$ that has attribute $X$ as $P(Y|X)$. This is just the conditional probability of $Y$ on evidence, or hypothesis, $X$.

A Bayesian Classifier then works as follows: during the training stage the classifier learns the posterior probabilities $P(Y|X)$ for each combination of $X$ and $Y$ based on

information gathered from the training data. Knowing these allows a test record $X'$ to be classified by finding the class $Y'$ that maximises the posterior property $P(Y'|X')$.

Naively, it would be very difficult to accurately estimate the posterior probabilities for every combination of class label $Y$ and attribute value, as a very large training set would be required even where the number of attributes is fairly small. However, the posterior probability can be expressed in terms of the prior probability $P(Y)$, the class-conditional probability $P(X|Y)$, and the evidence $P(X)$, via Bayes' Theorem viz.

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{4.1}$$

When comparing posterior probabilities for different values of $Y$, the denominator, being constant, can be ignored, and $P(Y)$ can be estimated from the training set by calculating what fraction of records belong to each class. It then remains to estimate $P(X|Y)$. This classifier is called 'naive' because it assumes that all the attributes are *conditionally independent*. Given this assumption, instead of computing the class-conditional probability for every combination of $X$, it is only necessary to estimate the conditional probability of each $X_i \in X$ given $Y$ – a task which does not require so large a training set. The classification process is performed by calculating

$$P(Y|X) = \frac{P(Y)\prod_{i=1}^{d} P(X_i|Y)}{P(X)} \tag{4.2}$$

and then choosing the class $Y$ which maximises the LHS, i.e. numerator of the RHS.

Naive Bayesian classifiers are good at handling isolated noise points, missing values, and irrelevant attributes, but their performance can be degraded by any correlations existing between attributes (since the independence assumption would not hold for such attributes – [Tan *et al.*, 2006] p237).

### 4.2.3 Artificial Neural Networks

"Neurologists have discovered that the human brain learns by changing the strength of the synaptic connection between neurons upon repeated stimulation by the same impulse. Analogous to human brain structure, an ANN is composed of an interconnected assembly of nodes and direct links." – excerpt from [Tan *et al.*, 2006] on neural networks, p246

Feed-forward neural networks trained by some variation of back-propagation have been used successfully in many pattern recognition problems. Essentially such a network provides a smooth function with bounded partial derivatives from input to output. It was shown by [Hornik *et al.*, 1989] and independently by [Cybenko, 1989] that by an appropriate choice of weights and thresholds, over a closed-bounded input set, a neural network can approximate with arbitrary precision any continuous function, i.e. neural networks are universal approximators. When used for pattern classifications the ability of neural networks to employ hidden units as a mechanism for constructing non-linear boundaries in pattern space is what makes them effective. However, as the continuous function approximation results might suggest, mapping pattern space into discrete regions is not truly the natural task of a neural network. In particular discrimination of inputs consisting of many discrete attributes into a small number of categories is a task more perhaps more naturally suited to decision-tree classifiers.

Of course, this is not to say that neural networks cannot perform such tasks, but rather that in such cases the *pre-processing* of the data so that *relevant* information is extracted and presented to the neural network inputs in a compact form becomes the critical aspect of the whole endeavour. This is even more the case when attempting to classify behaviours which are evolving over time. Here the best approach to encoding data for a fixed (hopefully relatively small) number of inputs is far from clear. Nor is it clear that a direct approach using feed-forward neural networks is necessarily appropriate.

Thus it would appear that in an attempt to use neural networks for this type of problem most of the work would go into determining an appropriate encoding. Indeed, so much so that one might suspect that one would have to essentially *solve the problem* before deciding on an appropriate encoding into input data suitable for a neural network discriminator.

For these reasons it was decided not to use neural networks as a classification method.

## 4.3 Classifier Implementations: Machine Learning Systems

It was decided, again at an early stage, that the aim should be to use a pre-existing machine learning system rather than develop a new one, which would have been a

difficult wheel to re-invent within the confines of a 3-year PhD. This section lists the machine learning systems used.

### 4.3.1 C4.5

C4.5 is a decision tree-based learning system written by J. Ross Quinlan [Quinlan, 1993a] in the early 1990s. Quinlan based C4.5 on ID3, a similar system he had developed previously. He has since developed various other machine learning systems such as FOIL and C5.0, and now runs a company called RuleQuest[2] which sells data mining software. The source code of C4.5 is still made available for download, albeit with a rather onerous set of restrictions on distribution. It was originally designed for use on a UNIX environment, but with the help of MINGW[3] and CYGWIN,[4] a native Windows port was quite easy to produce.

C4.5 takes its input from a series of files, all of which must have the same 'filestem' (the part of the filename before the dot). Assume for the sake of this example that the filestem is 'ExperimentOne'. The attributes are defined in a file called 'ExperimentOne.names', whereas the training data must be stored in 'ExperimentOne.data' and the test data in 'ExperimentOne.test'. The data and test files contain only the data values in comma-separated format.

The output files produced by C4.5 have the same filestem, using different extensions to distinguish between them. Running C4.5 on 'ExperimentOne' would produce a decision tree output file called 'ExperimentOne.tree' and an unpruned tree output file called 'ExperimentOne.unpruned'. If the ruleset generator (C4.5Rules) was then run, it would produce an output file called 'ExperimentOne.rules'.

The table below shows an example of a names file. The first line indicates the classes. Then, individual attributes are defined in the form 'attribute-name: type.' where attribute-name is a string naming the attribute and type is a C4.5 keyword indicating the attribute type. Attribute types may be as follows:

1. **ignore** – an attribute not to be used for classification. Useful for naming cases.
2. **continuous** – a continuous quantity.

[2]http://www.rulequest.com
[3]http://www.mingw.org
[4]http://www.cygwin.org

3. **discrete N** – specifies that attribute has N discrete values, N being a positive integer.

4. comma-separated list – like "discrete", but specifies the items explicitly.

Table 4.1 shows an example names and data files for a data set which might be used to see whether it is possible to determine which of three political parties an individual is likely to vote for in an election based on various attributes such as their age, marital status and income (of course, the data given here was invented at random and no inferences should be drawn from it). In practice, such data would never include personally identifiable attributes such as a voter's name – this would be replaced by a neutral identifier.

| **NAMES FILE** |
|---|
| Conservative, Labour, Liberal Democrat. |
| |
| Name: ignore. |
| Age: continuous. |
| Gender: Male,Female. |
| AnnualIncome: continuous. |
| MaritalStatus: Single,Cohabiting,Married,CivilPartnered,Divorced,Widowed,Other. |
| NumberOfChildren: continuous. |
| Religion: None,Christian,Jewish,Muslim,Hindu,Buddhist,Other. |

| **DATA FILE** |
|---|
| Voter 1, 35, Male, 15000, Married, 2, None, Conservative |
| Voter 2, 28, Female, 25000, Single, 0, Christian, Labour |
| Voter 3, 45, Male, 75000, Divorced, 2, Christian, Conservative |
| Voter 4, 30, Female, 30000, Married, 2, Muslim, Conservative |
| Voter 5, 50, Male, 40000, CivilPartnered, 1, Jewish, Labour |
| Voter 6, 78, Female, 2000, Widowed, ?, Buddhist, Liberal Democrat |
| ... |

Table 4.1: Example C4.5 Input Data Files

In the data file, question marks may be used to indicate unknown attributes (this is also true of WEKA). This can be seen in the case of Voter 6 in Table 4.1. This allows cases to be classified where not all the attributes are necessarily known.

The C4.5 distribution includes various tools beside the basic decision tree generation program (*C4.5*). There is also a ruleset generator (*C4.5rules*), a program for interactively classifying cases against an existing model (*consult* and *consultr*, and a script for performing cross-validation. Cross-validation works by dividing the input data into $n$ blocks. This is done in such a way that the number of cases in each block and the class distribution within it is uniform. Then, $n$ different models are built by omitting

one block at a time, building a model on the remaining data and testing it against the omitted block (thus, each case appears in exactly one test set). If $n$ is not too small, the average error rate over the $n$ blocks is a good predictor of a model built from all the data ([Quinlan, 1993b] p89–90 and [Tan *et al.*, 2006] p187). WEKA also supports cross-validation.

C4.5 only supports one classification method, and the range of output statistics it gives is neither as extensive nor as well-documented as WEKA. For this reason, and because WEKA supports J4.8 (essentially the same classification algorithm as C4.5), it was eventually decided to use WEKA as the main learning system. However, C4.5 was used in the early structural experiments before replicating the results with WEKA, which provided a means of verifying that the Windows port of C4.5 was working correctly.

### 4.3.2 WEKA

The WEKA machine learning toolkit was developed at the University of Waikato in New Zealand and described in [Witten and Frank, 2005]. This provides not only a number of decision tree classifiers (including J4.8 which is functionally equivalent to C4.5), but also a vast number of other classification methods, including Naive Bayes.WEKA is written in Java, making it cross-platform, and is free software licensed under the GNU General Public License – the version used here is 3.4.10.

Data are input to WEKA in the form of 'ARFF' files. Unlike C4.5, the attribute information and the data are stored in the same file. Data are comma-separated and terminated with a newline character. Asterisks are used to indicate keywords, and comments are indicated by the percent sign. The ARFF format includes support for string attributes, but the classifiers used in this work do not support using strings for classification. This, combined with the fact that there is no provision to specify an attribute as 'ignored' (which C4.5 can do), means that the identifiers for individual cases must either be removed completely (an annoyance, since it is no longer possible to identify individual cases in the data file) or put in as comments. The latter workaround is illustrated in Table 4.2, which gives a sample WEKA input file for the same data set as was used to illustrate C4.5 in Table 4.1 above.

The only disadvantage WEKA has is that it does not support "ignoring" attributes as did C4.5 – each attribute must be used for classification. Thus, individual instances cannot in practice be named as in Table 4.2. The work-around for this is to use comments to name the individual instances.

| AN EXAMPLE WEKA ARFF FILE |
|---|
| % lines beginning with a percent are comments |
| |
| % name of relation @relation Voters |
| |
| % attribute definitions start here |
| @attribute Age integer |
| @attribute Gender {Male,Female} |
| @attribute AnnualIncome integer |
| @attribute MaritalStatus {Single,Cohabiting,Married,CivilPartnered,Divorced,Other} |
| @attribute NumberOfChildren integer |
| @attribute Religion {None,Christian,Jewish,Muslim,Hindu,Buddhist,Other} |
| |
| % the last attribute gives the class |
| @attribute PartyVotedFor {Conservative,Labour,Liberal Democrat} |
| % the data starts here |
| |
| @data |
| % Voter 1 |
| 35, Male, 15000, Married, 2, None, Conservative |
| %Voter 2 |
| 28, Female, 25000, Single, 0, Christian, Labour |
| %Voter 3 |
| 45, Male, 75000, Divorced, 2, Christian, Conservative |
| %Voter 4 |
| 30, Female, 30000, Married, 2, Muslim, Conservative |
| %Voter 5 |
| 50, Male, 40000, CivilPartnered, 1, Jewish, Labour |
| %Voter 6 |
| 78, Female, 2000, Widowed, 4, Buddhist, Liberal Democrat |
| ... |

Table 4.2: Example WEKA ARFF File

WEKA classifiers can be invoked from the command line, but there is also various GUI-based interfaces (in order of complexity these are Weka Experimenter, Weka Explorer and Weka KnowledgeFlow) which provide greater ease of use and a considerable amount of extra functionality. However, it is arguable that the capabilities of WEKA were severely under-utilised in this PhD, as little of the extra functionality was used. In practice, the command-line was used for the structural classification experiments described in Chapter 6 but the Weka Explorer was used for the behavioural classification experiments described in Chapter 7.

## 4.4 Conclusion

To sum up, it was decided to use only the decision tree and Naive Bayes classifiers. As to implementation, the original intent for the structural experiments was to use

C4.5 as the decision tree classification system and WEKA for Naive Bayes, but it was eventually decided to use WEKA only since it supports both classifiers, retaining the C4.5 results for comparison purposes but not actually using them. The behavioural experiments used WEKA only.

Of course, having chosen the classification system, it is necessary to collect data in a format appropriate to it. Chapter 5 will describe the methods used to collect this data.

Chapter 5

# Gathering structural and behavioural data in Windows

## 5.1 Introduction

This chapter provides an explanation of the techniques used to gather both *structural* and *behavioural* data. Throughout this thesis, "structural data" refers to data derived from the structure of an executable disk file,[1] whereas "behavioural data" refers broadly to data collected from a running process which may change depending on the operations which the process is carrying out.

To classify malware it is necessary to collect both structural and behavioural data from static analysis and real-time monitoring. When considering the types of data which it is feasible to collect it is important to recognise that there are several constraints on the freedom of choice. These are primarily what can be done within the operating system, but also relate to the processor overhead; the monitoring process should not be so demanding that it significantly degrades the performance of the overall system from a user perspective.

Under Windows, when a program is running (loaded into memory and executing) it is referred to as a 'process'. Since a program must be running to be 'behaving' then it is only appropriate to speak of 'gathering behavioural data' from a running process. Structural data can be gathered either from a running process or from a program file on disk. However, for these purposes structural data is always obtained from a disk file and not from a program in memory. Leaving aside the abstractions (such as

[1] This use of the term "structural" should not be confused with the definition used in the field of software structure metrics

process and thread objects) which Windows uses, the structure of the memory image for a normal program does not usually differ much from the structure of its associated disk file. However, a virus is not a normal program, and as such can have a very different structure when it is executing in memory to when it is stored as a program file (particularly if it is encrypted or poly/metamorphic).

## 5.2 Structural Data

For structural classification, it should be possible to predict what sort of data the learning system ought to select, based on ideas already developed by virus researchers. If the attributes picked out by the learning system, particularly for viruses, were to tally with those given by Szor (see Table 5.1) or other researchers, this would be good evidence that the learning system is working as intended. It would also subsequently be possible to think up hypothetical significant factors and see if the learning system confirms or disconfirms their significance.

### 5.2.1 What sort of data to use?

In the case of a file-infecting virus, there are a number of structural features which an expert would expect to occur more commonly in infected programs than in uninfected programs. In [Szor, 2005] a list of such features is given, and some of these are reproduced in Table 5.1.

It would be very simple to add facilities to a scanner program to detect these anomalous features, and indeed many virus scanners do use some form of heuristic detection in addition to the signature process. However, it is not possible to rely on Szor's list (or any such list) as *definitive* given the high rate of malicious code evolution. Therefore, data must be collected that would allow a learning system to pick out the features that *it* finds are significant, which may or may not correspond to lists produced by human experts.

### 5.2.2 The Portable Executable (PE) file format

The format Windows uses for executable files is known as the Portable Executable format. The PE format is used for programs (*.exe or *.scr), DLLs (*.dll) and device

drivers (*.sys), and also for object files produced by Microsoft compilers. It is based on the older Common Object File Format, and retains a degree of similarity to COFF. Microsoft's documentation for the PE format has been extensively updated since the format was first introduced (it replaced the NE format used by 16-bit Windows, which in turn was a replacement for the MS-DOS EXE format). Microsoft's current (August 2006) official documentation for the PE format [Microsoft, 2006] runs to 73 pages, but for those who do not need to know about the latest features a highly accessible guide is given by Matt Pietrek in the February and March 2002 issues of MSDN Magazine [Pietrek, 2002]. These articles were based on Pietrek's earlier article 'Peering inside the PE' [Pietrek, 1994 updated 2002], originally published in the Microsoft Systems Journal.

Table 5.2 shows the structure of a PE file. As with every other executable file since the days of MS-DOS, a PE file begins with a MZ header. Assuming MS-DOS is not in use, only one field in the MZ header 'matters' – the field at address 0x3C – called 'e_lfanew' by Microsoft. This is a pointer to the PE file header. The 'DOS stub' is a tiny MS-DOS program which simply prints out the message "This program cannot be run in DOS mode." and then exits. The idea of this is to provide a sensible error message if someone tried to run the program in a pure DOS environment – since DOS would ignore the e_lfanew field, the stub would be executed instead.

| Structural features characteristic of an infected program file |
|---|
| Code execution begins in last section |
| Suspicious section characteristics, names, or flags |
| Virtual size incorrect in Portable Executable (PE) header, or incorrect sizes in other headers |
| Suspicious gaps in sections |
| Abnormal jumps to other sections |
| Entry point in PE header points somewhere other than the .text section |
| DLL imports by ordinal |
| Patched Import Address Table (IAT) |
| Multiple file headers |
| CALL to POP, or CALL to next instruction |
| Bad checksums in DLLs |

Table 5.1: Some of the structural anomalies common in virus-infected program files as described by Szor

| DOS MZ Header |
|---|
| DOS Stub |
| PE Header |
| Section Table |
| Section 1 |
| Section 2 |
| Section .. |
| Section N |

Table 5.2: Diagram of PE File Structure

### 5.2.3 The PE Header

After the DOS stub comes the PE file header, identifiable by its 4-byte signature consisting of the characters 'P' and 'E' followed by two null bytes. The PE file header contains two parts: a COFF file header containing 7 fields, and an optional header whose length may vary. The optional header contains 9 standard COFF fields (or 8 in PE32+), 21 further Windows-specific fields and a data 'directory'. The data directory consists of a set of structures containing a 32-bit address field and a 32-bit size field. The number of such structures is specified in the last Windows-specific field (NumberOfRvaAndSizes). The data directory is used to store addresses of various important parts of the program file (such as the import and export tables).

### 5.2.4 Sections and the Section Table

After the optional header comes the section table. An interesting point about the section table is that its address is not stored in any of the headers. Instead, it must be calculated by working out where the optional header ends (using the SizeOfOptionalHeader field in the COFF file header). Each entry in the section table contains 10 fields describing a particular section, covering such information as the section's name, its virtual address and size, and whether it contains any relocations.

Sections are the basic 'building block' used when the PE file is being created by the linker. They are roughly analogous to 'segments' in older executable formats. Normal programs have one section – usually called '.text' or '.code' – which contains all the code. Other sections contain various types of data, resources, or the addresses of imported or exported functions. The Windows loader limits a program to 96 sections. Sections have flags (the 'Characteristics' field of the section table entry) which mark

whether they are readable, writable, or executable. Attempting to write to a read-only section results in a page fault. Some sections are mapped into memory from the program file by the loader, whereas others are created by the loader and do not physically exist in the program file until it is mapped into memory prior to execution.

### 5.2.5 Viruses and the PE format

PE-infecting viruses must clearly be able to process all the above information in order to infect successfully. Viruses must read the headers and section tables of the file they wish to infect in order to gather information. Furthermore, they must modify these headers as well. Since the modifications made by a virus are unlikely to exist in an uninfected file, they can be said to be 'characteristic'. For example, Peter Szor's list of anomalies (see Table 2.2), which will now be explained in the context of PE infecting viruses. It should be noted that these anomalies apply mainly to viruses only, though any that apply to other types of malware will be noted.

#### Code execution begins in last section

One common infection strategy is to add the virus code as an extra section. The easiest place to add an extra section is at the end of the host file. In some cases, there is enough empty space in the current last section to insert the virus code without needing to add a new section. But whichever of these strategies is used, the virus code must be made to execute before that of the main program. The easiest way of doing this is to modify the entry point in the host file's PE header, but to make detection more difficult, a trick known as Entry Point Obscuring can be used. The simplest method is to modify the first few instructions at the existing entry point instead of modifying the header (of course, if this is done it is then necessary to save the instructions that have been modified and append them to the end of the virus code so that the program still works). If properly done, this can mean that it is more difficult to detect that execution begins in the last section, though the fact that there is a jump to the last section shortly after the beginning of the program ought to be suspicious in itself (see below).

#### Suspicious section characteristics, names, or flags

Although theoretically sections can have any name, the majority of noninfected programs have a small set of common section names, such as '.text', '.code', '.bss', '.rsrc',

'.reloc', '.edata', or '.idata'. These reflect conventions used by the majority of linkers. Therefore, the presence of sections with unusual names can be cause for suspicion. Normal programs that have unusual sections include programs protected by some types of copy protection as well as programs that have been compressed with executable "packers" such as UPX (see Section 5.3.1), and these may generate false positives, particularly as many malicious programs are UPX-compressed in an attempt to make them smaller and more difficult to detect. If a virus inserts its code into a section which does not already have its execute flag set, the virus must modify the section flag in order to allow execution. Thus, a section which should not need to be executable but which has the execute flag set is suspicious. However, not all viruses need to do this since in many environments code may be executed in any section marked as writable even if the executable flag is not specifically set. Both hardware and software technologies are currently in use to prevent execution in non-code memory pages (an example being Microsoft's Data Execution Prevention system), but these are designed to prevent buffer overflow exploits and are more useful as a weapon against attack from outside by worms than against malware already present on a system.

### Virtual size incorrect in PE header, or incorrect sizes in other headers

A virus will generally make the minimum number of modifications to a new host it can get away with to allow its code to run. The Windows loader is known to depend on certain values in the header of a program and to ignore others, meaning that viruses do not always change them to reflect the modifications they have made to the host file (see [Szor, 2005] p164). Thus, a discrepancy between what these values are and what they should be can indicate infection. Two examples given by Szor are the SizeOfCode and the SizeOfImage fields from the optional header. The SizeOfCode field should contain the (rounded) sum of the sizes of all the executable sections in the program file. Many viruses, according to Szor, do not fix this value after adding more code to the host, so a discrepancy between the header value and the actual sizes of the sections could indicate infection. The SizeOfImage field was not checked by the Windows 95 loader, so many Windows 95 PE viruses did not bother to alter this field. The NT loader, however, did and does check this value, meaning that modern viruses must recalculate it properly in order to spread on modern Windows systems.

### Suspicious gaps between sections

These can occur where a virus which adds a new section realigns the file before adding the new section, but fails to alter the section table for the former last section to take this into account.

### Abnormal jumps to other sections

The only jumps from the main code section to other sections that a normal program should possess are the jumps to the table of imported functions (see below). Any other jumps to other sections are possible indicators that code has been inserted.

### Entry point in PE header points somewhere other than the '.text' section

As has been previously mentioned, when the linker builds a PE executable, all the code is put in one section, called '.text' or '.code'. The AddressOfEntryPoint field in the PE header is a pointer to the place where execution begins, e.g. to the first machine instruction of the program. There are very few occasions where the entry point should be outside the '.text' section, and a program file having an entry point in the last section is extremely suspect (see Section 5.2.5, but also see Section 5.3.1).

### DLL imports by ordinal

Imported functions are functions (such as the Win32 API functions) which reside externally to the program. They are usually located in dynamically-linked libraries (DLLs). The most common method by which a program uses functions in DLLs is called 'implicit linking'. The '.idata' section of a program file contains various tables of information about the functions it imports. When preparing a program for execution, the loader, having mapped all the necessary DLLs into the address space of the program (which has now become a *process*), replaces some of the .idata information with the actual addresses of each function.

Without going into details of how this is achieved (for these see [Pietrek, 2002]), each imported function has associated with it a value known as the *ordinal*, which is used by the loader to quickly find the function in the DLL. The table of functions imported from a particular DLL contains, in each entry, the name of the function (e.g. MessageBoxA) and its likely ordinal value. This ordinal value is only to help the loader find the

function and need not actually be correct (which is why the ordinal field is called "Hint" in the Microsoft documentation). However, it is possible (but rare) to import a function 'by ordinal' rather than using the function name.

One of the problems a virus has is getting the addresses of API functions it needs. The virus does not know where in the host program its code will end up, nor what functions the host program will import. Some viruses will be capable of reading the host's import tables and getting the addresses of its functions from there – but if the virus needs a function which the host doesn't already import, it will have to either patch the host's import table or explicitly import the function using GetProcAddress(), an API function for finding the address of a function in a DLL. GetProcAddress() can take either a function name or its ordinal as an argument.[2] Whichever approach is taken, it is much easier and more compact to store an ordinal (being an integer) than to store a function name (which is a string) in virus code, and furthermore strings of function names are easily detectable. Hence the incentive to use imports by ordinal. Unfortunately, ordinals are not constant between versions of a DLL – they may change if new functions are added. Thus, if only the ordinal is provided to GetProcAddress() it might just return the address of the wrong function. This meant that viruses which made use of imports by ordinal would often 'break' when introduced to new builds or versions of Windows.

### Patched Import Address Table (IAT)

Some viruses will go ahead and patch the import table (or explicitly import functions) without checking to see if the functions are there already. Thus, if a set of functions is imported twice (once by name and once by ordinal) or if a set of functions already in the IAT is imported explicitly somewhere else using GetProcAddress(), it's a good sign that something untoward is going on.

### Multiple file headers

Many of the items in this list, whilst characteristic of infected files, are not unique to them and could also be found in a small number of uninfected files (for example, programs which use certain forms of copy-protection or compression). It is almost impossible to conceive of a normal reason for having multiple file headers in a file, but

[2] But how to find GetProcAddress()? In early days its address was hard-coded, resulting in viruses which would only work on specific versions and builds of Windows.

this is in fact characteristic of very basic prepending viruses. These viruses do not 'infect' the host in the usual sense, but rather 'engulf' it. This is done by creating a new program file containing the virus code, and copying the host program to the end of this file, complete with all headers. The original host file is then deleted, and the new file given the same name. When run, the virus code executes. When it has finished, it copies the old host program into a temporary file and executes it. The host program's code is never actually modified.

This trivial infection method is relatively simple to program (thus most such viruses are written by beginners and in high-level languages) but is also relatively simple to detect, so refinements have been developed ([Szor, 2005] p 133).

### Relative CALLs with zero offset, or CALLs into the middle of another instruction

There are a number of tricks which allow a piece of relocatable code to get its own address, a particular concern for viruses and shellcode. The simplest version is known as 'CALL-to-POP'. The trick exploits the fact that when a CALL instruction is executed the address of the next instruction is pushed onto the stack, which allows the processor to return control to that location after executing a RET instruction.

| Machine Code | Assembly |
|---|---|
| E8 00 00 00 00 | CALL next |
| 58 | next: POP EAX |

Table 5.3: Call-to-Pop: Simplest form

Table 5.3 shows the simplest form of the CALL-to-POP trick – and the easiest to detect, since there is no conceivable need to have this combination of instructions in a normal program. Most shellcode uses a slightly different method, first popularised in [AlephOne, 1996], where the first instruction is a JMP pointing to the beginning of the string data area, where there is a CALL instruction pointing back up to the next instruction after the JMP. The address of the first character of the string will be pushed onto the stack before the CALL is executed, and can be POPed off and stored.

Table 5.4 gives AlephOne's shellcode. This is not Win32 code (it makes Linux system calls using Interrupt 80) but it uses the same principle. Some of this code appears rather tortuous, but this is only because no instructions containing zero bytes are allowed, as shellcode has to look like a standard character string, which uses zero as a

| Machine Code | Assembly | Comment |
|---|---|---|
| EB 1F | JMP string-area | Go to CALL |
| 5E | first: POP ESI | Get string address off stack |
| 89 76 08 | MOV DWORD PTR[ESI+08],ESI | Store it in memory |
| 31 C0 | XOR EAX,EAX | Clear EAX |
| 88 46 07 | MOV BYTE PTR[ESI+07],AL | Store a null byte in memory |
| 89 46 0C | MOV DWORD PTR[ESI+0C],EAX | Store a null DWORD |
| B0 0B | MOV AL, 0B | Put value 0B in AL |
| 89 F3 | MOV EBX, ESI | The address of the structure |
| 8D 4E 08 | LEA ECX, DWORD PTR[ESI+08] | Address of address |
| 8D 56 0C | LEA EDX, DWORD PTR[ESI+0C] | Address of null DWORD |
| CD 80 | INT 80H | Call the kernel |
| 31 DB | XOR EBX, EBX | Zero out EBX |
| 89 D8 | MOV EAX, EBX | Zero out EAX |
| 40 | INC EAX | Put 1 in EAX (system exit) |
| CD 80 | INT 80H | Call the kernel |
| E8 DC FF FF FF | string-area: CALL first | |
| ... | (start of string data) | |

Table 5.4: Call-to-Pop in AlephOne's Shellcode

terminator. It is thus necessary to use only those instructions for which the machine code contains no zero bytes.

There are a number of more advanced variations on the theme, most of which aim to make detection more difficult. One particularly ingenious method involves a CALL to a location in the middle of a block of garbage instructions which happens to contain the correct value for the POP opcode (or somehow obtains the address on the stack). The garbage instructions can be made to look realistic, but they are never executed and can therefore be complete rubbish if necessary.

| Machine Code | Assembly |
|---|---|
| E8 0A 00 00 00 | CALL ip+0x0A |
| FF 08 | DEC DWORD PTR[EAX] |
| 33 C0 | XOR EAX,EAX |
| 55 | PUSH EBP |
| 89 E5 | MOV EBP,ESP |
| FF 15 20 58 C3 00 | CALL DWORD PTR[0x00C35820] |
| XX XX XX XX XX | (data bytes) |

Table 5.5: Call-to-Pop: Covert

The code shown in Table 5.5 appears to be a call instruction followed by what looks like a 'function prologue' – the set of instructions which generally occur before a function call – and the function call itself. In fact, the first CALL instruction is pointing into the middle of the second CALL instruction, which just happens to contain the bytes 0x58

and 0xC0 – in other words, the instructions POP EAX and RET! The instructions after the first CALL are never executed and just serve as a diversion. This trivial example is still fairly easy to spot, but in practice there might be an indefinite number of dummy instructions between the CALL and the hidden POP instruction (of course, the code needs to know how many dummy bytes there are and add that number onto the value in EAX to get the address of the data area, but that is easy enough.

The reason for needing to know an absolute address is because injected code must often contain tables of string data of some sort (such as API function names) and these need to be referenced by address (unlike numerical values which may be hard-coded). Thus, some way of getting the address of the string data are necessary. One method would be to use relocations – recalculate and patch all the references to the string data when copying the virus code to a new host – but it is often simpler to write the code in such a way that it is fully relocatable and can obtain its own address.

Table 5.6 is a small patch which illustrates this process (note that this patch does require the API address).

| **Machine Code** | **Assembly** | **Comment** |
|---|---|---|
| 50 | PUSH EAX | Save any value originally in EAX |
| 33 C0 | XOR EAX,EAX | Get 0 in EAX |
| 50 | PUSH EAX | First argument (hWnd = NULL) |
| E8 1B 00 00 00 | CALL ip+0x1B | Get address of string area |
| 05 02 00 00 00 | ADD EAX,0x02 | Get address of message string |
| 50 | PUSH EAX | Second argument (message string) |
| 05 11 00 00 00 | ADD EAX,0x11 | Get address of caption string |
| 50 | PUSH EAX | Third argument (caption string) |
| 33 C0 | XOR EAX,EAX | Get 0 in EAX |
| 50 | PUSH EAX | Fourth argument (MB_OK) |
| FF 15 XX XX XX XX | CALL MessageBoxA | (address must be substituted for XXs) |
| 58 | POP EAX | Restore original value in EAX |
| E9 XX XX XX XX | JMP OriginalEntryPoint | (address must be substituted for XXs) |
| E8 00 00 00 00 | CALL ip+0 | Do Call-to-Pop trick |
| 58 | POP EAX | Get address in EAX |
| C3 | RET | Return to caller |
| 54 68 69 73 .... | "This .." | Beginning of string area |

Table 5.6: Call-to-Pop in use

The patch code in Table 5.6 is from a small program which adds a patch to an existing program to display a message box with the text "This is a patch.". Like many viruses, the program inserts this patch into existing space in the program and alters the AddressOfEntryPoint field of the header to point to the patch code. When the patch code finishes it jumps back to the original entry point address, so that after the patch code is executed the original program runs as normal. The program which inserts the patch

searches the host program's import table for the MessageBox function and adds this to the patch in the correct place before insertion. The patch makes use of the Call-to-Pop trick to get the address of the strings to be displayed in the message box.

#### Bad checksums in DLLs

This is somewhat obsolete as a suspicious feature, since all current versions of Windows require DLLs and system programs to have a valid checksum, and modern viruses are in any case quite capable of recalculating the checksum. However, older versions of Windows ignored the checksum completely, with the result that viruses which infected DLLs rarely bothered to recalculate the checksum. DLL infection in current versions of Windows is made more difficult again, at least for any files Microsoft deems critical, by the System File Protection feature. This maintains a cache of backup copies for all these files, detects any attempt to delete or modify them, and promptly restores the backup copy when this happens. More recent versions of Windows support cryptographic signatures instead of checksums.

That being said, bad checksums in program files are still a potential sign of infection. Historically, most programs that are not 'system programs' have had their checksum set to zero by the linker, which is fine as the Windows loader did not make use of this field (only 'system programs' and DLLs need valid checksums). Until very recently, viruses could therefore use the Checksum field as an 'infection marker'. Infection markers are used by viruses to identify already-infected files and thus avoid both wasted effort and any problems which may be caused by infecting the same file twice. Thus, a program file with a Checksum value which is nonzero and is not a correct checksum of the file is automatically suspect.[3] It should be noted that current versions of Windows (XP, Vista) are much more stringent about executable verification.

### 5.2.6 Collecting Structural Data

Having seen what types of structural data might be indicative of infection, the question arises as to how such data can be collected. Most of the data discussed above may be collected by simply mapping a program file into memory and accessing its headers. Some data (such as the number of imported functions) may require accesses to the

[3] It is also conceivable, but unlikely, that a covert virus might set a correct checksum as an infection marker, making detection more difficult and giving the beneficial side-effect of stopping the virus making its presence too obvious by infecting system files.

section data and a certain amount of calculation, and finding suspicious redirections or tricks like CALL-to-POP requires in-depth code analysis, the automation of which is beyond the scope of this project. The items of structural data which were eventually chosen and the method used to collect them are set out in Chapter 6.

## 5.3 Behavioural Monitoring

'Behavioural Monitoring' is defined to be monitoring a process during execution, looking at how it uses memory, the CPU, and other resources, whether it opens files, starts child processes, and any similar 'behaviour'. Rather than define a list of 'behaviours', the aforementioned measurements are 'sampled' at specified intervals over a given time period.

### 5.3.1 The collection of real-time process data

The range and type of data to be used for the characterisation of processes will obviously be operating system dependent. Process *behaviour* is essentially the evolution of the descriptive statistics over time. For *structural* data mentioned in Table 5.1, originally published in [Szor, 2005] mostly in relation to viruses, a list of characteristics which might be expected to help discriminate malware.

For *behavioural* characteristics as far as can be determined no-one has ventured to publish an equivalent list. So in Table 5.7 some initial suggestions are given.

Each item in Table 5.7 will now be explained, using examples from real malicious programs where possible. The first example program is Backdoor.Win32.Asylum, a typical remote-control program. Although Asylum is quite an old example (circa 2000) it contains enough pertinent features to be a useful example. The variant used here – denoted 012 – is credited to 'slim', though since this backdoor was made 'open-source' it is evident that other authors have contributed to the many variant versions.[4] However, the binary file server variant examined had clearly been modified, as it contained features not present in the source code (assembly) version. Like most such programs, Asylum consists of a small 'server' program, written in Assembler, which, when executed, copies itself to the Windows system directory and writes itself into the Registry so that it is

[4] After spending a day disassembling the server program and reverse-engineering it, it was discovered that the full source code was only a web-search away...

| 'Suspicious' behaviours |
|---|
| Opening outward internet connections |
| Writing to executable files |
| Self-modification |
| Writing to files normally created by other standard programs |
| Spawning multiple processes (in particular functional replication) |
| Disinclination to be terminated |
| Altering permissions |
| Writing to system areas |
| Attempting to capture data from other programs |
| Writing to specific Registry keys |
| Suspiciously high resource usage |

Table 5.7: Some of the behavioural properties that might be expected to help detect malware.

executed whenever Windows starts up. It then listens on a port (default 23432) for incoming connections, and may also start a separate thread which attempts to post a notification message via ICQ. The message says "hey there, ive(sic) been committed" and gives details of the infected computer's hostname, IP address, the port number of the Asylum server, and various other information. Whether or not it does this depends on how the server program has been configured.

An attacker can now point the 'client' program (see screenshot below) at the infected computer and connect to the server. There are ten different commands the client can send to the server, though six of these are for connection-control purposes only. Once connected via the client, an attacker can do four different things: upload an arbitrary file, execute an arbitrary program, reboot the computer, and remove the server program. The client program is written in Borland Delphi, and the example seen here was also compressed using UPX.

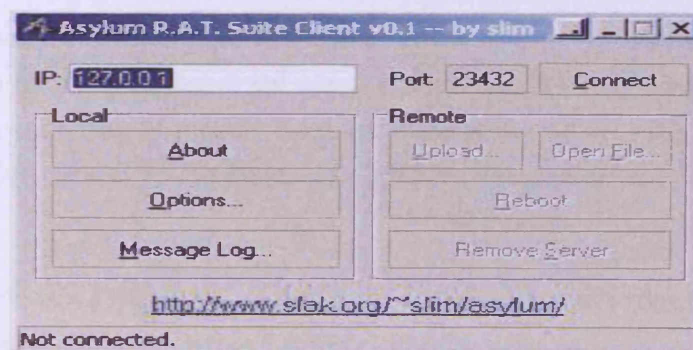



Figure 5.1: The Asylum backdoor client program by 'Slim'

As well as the client program, there is also a configuration program which allows the server binary to be customised, allowing the attacker to select the manner in which the server stays resident on the infected computer, and whether or not the server should announce its presence via ICQ. The configuration program, which is also written in Borland Delphi, configures the server by modifying the binary directly, writing to locations which are then checked in the server code. To illustrate features not found in Asylum, a second example will be used: the Win32.Welchia worm. This worm, which spread by exploiting a buffer overflow vulnerability in the Windows RPC service (though it could also use a different exploit), is interesting as an example of 'helpful malware' because, once it infected a system, it would attempt to download and install Microsoft security patches to block the vulnerability by which it had entered the system. If it found an instance of another worm (Win32.Lovsan/Win32.Blaster) on the system, it would attempt to remove it. It would also automatically remove itself after 190 days or 1st January 2004, whichever occurred sooner. However, not all features of Welchia are benign [Perriot, 2007].

These examples will now be used to illustrate the list of suspicious behaviours given earlier.

### Opening outward internet connections

Nearly all malware apart from 'pure' file-infecting viruses will need some sort of communication facility. In some cases this is to allow them to 'call home' and let attackers know of their presence, or to allow attackers to communicate with them, or to send stolen data. Generally, therefore, they will need to gain access to the network somehow. Asylum, like many malicious programs, uses the Winsock API functions to gain access to the Windows TCP/IP stack. When the server starts, it attempts to bind to and listen on (by default) port 23432. If it is unable to perform any of these steps, it puts its main thread to sleep for 512 ms and tries again. As TCP is a transport-layer protocol, it is necessary to define a higher-level communications protocol on top. Asylum uses a simple protocol consisting of mostly trigram commands: RQS, PAS, GNT, DNY, DIE, UPL, RBT, WDR, SDR, RUNCMD, and INV to denote an invalid or unrecognised command.

However, Winsock is not the only communication option open to a malicious program. Microsoft has helpfully provided a number of API wrappers for higher-level communications protocols. In particular, the Windows Internet functions in WININET.DLL can handle HTTP, FTP and GOPHER transactions with a remote site, and thus a

| Asylum Commands | |
|---|---|
| RQS | Request connection |
| PAS | Password |
| GNT | Granted |
| DNY | Denied |
| DIE | Terminate server app |
| UPL | Upload file |
| RBT | Reboot machine |
| WDR | Windows directory |
| SDR | System directory |
| RUNCMD | Run file |
| INV | Invalid command |

Table 5.8: The commands used by Asylum's communication protocol

malicious program which restricts itself to these protocols can communicate without the hassle of implementing a protocol on top of TCP or UDP.

Certain types of malware, such as worms, are known to exhibit characteristic patterns in network traffic (see graphs in [Szor, 2005] p317) which may aid their detection. However, detecting such patterns is somewhat beyond the scope of this PhD and would be a more appropriate task for a firewall or Intrusion Detection System (but see the section on suspicious resource usage below).

### Writing to executable files, or files normally created by other standard programs

Very few programs need to write to pre-existing executable files. Even programs which create executable files (compilers and assemblers) usually create a new one every time. Thus, writing to an already-existing executable file is a suspicious activity. So is attempting to replace an executable file which is part of the operating system, though Windows now has built-in (but not foolproof) protections against this.

The Asylum customiser program writes directly to the server binary in order to do its customisation, modifying locations which are then checked when the server runs to determine its behaviour. In a nonmalicious context, software vendors sometimes release binary update patches which modify executable files, though due to increases in bandwidth and storage space it is now more common to replace the entire file with the upgraded version. The only other programs which might legitimately write to existing binaries are programs such as hex editors which the majority of users would

probably never have to deal with.

#### Self-modification, or a program which accesses its own executable file

'Self-modification' refers to a process performing manipulations on its own image in memory. This commonly indicates a copy-protection scheme, a polymorphic or metamorphic virus, or a program that has been packed with a program compression/encryption utility. Malicious programs are often so treated, as the process has two advantages – it makes the malware executable smaller and easier to transmit (and hide), and it also obfuscates the code to a greater or lesser extent, making it somewhat more difficult for anti-malware researchers or programs to analyse. The packing program compresses and sometimes encrypts the target's code. When the compressed program is run, a stub decompresses/decrypts the data, writes it to memory, and passes execution to it. Many commercial packers are available, as is an open-source one called UPX.[5] A notable feature of UPX is that its authors sensibly refuse to add a feature to prevent unpacking, arguing that such features are trivial to bypass and provide a false sense of security. The commercial packers, on the other hand, often use encryption and anti-reverse-engineering features as selling points. Some packers also polymorph their unpacking code, making different generations of the same malicious program appear dissimilar.

The Asylum client and configuration programs are distributed as UPX packed executables, but the Asylum server is not. As anyone who possesses the UPX program can unpack these files, the motive for packing appears to be purely to make the executables smaller. The Welchia worm is distributed as a packed executable. Some sources suggest that UPX was used – however, the standard version of UPX is not able to unpack Welchia. is not difficult to trace the unpacking routine in a debugger, and by setting a breakpoint in the appropriate location it is possible to view the unpacked code in memory. Table 5.9 shows the first and last instructions of Welchia's unpack stub.

The packed executable, when mapped into memory, contains three sections. One contains the packed code, together with the unpacking stub, and starts at address 0x00407000. The loader creates an identically-sized section at 0x00401000 which will eventually contain the unpacked program. The ESI and EDI registers are loaded with the address of the source and destination areas respectively, and the unpacking process commences. After the process is finished the registers are restored and control is transferred to the entry point of the unpacked program (0x00402FCC). More advanced

[5] http://upx.sourceforge.net

| Machine Code | Assembly | Comment |
|---|---|---|
| 60 | PUSHAD | Save registers on stack |
| BE 00 70 40 00 | MOV ESI, 00407000 | Address of packed data |
| 8D BE 00 A0 FF FF | LEA EDI, DWORD PTR[ESI+FFFFA000] | Address of unpacking area |
| 57 | PUSH EDI | EDI saved onto stack |
| 83 CD FF | OR EBP, FFFFFFFF | Set EBP to FFFFFFFF |
| EB 10 | JMP 00409082 | Go to next part of unpack fn |
| (...) | (...) | (...) |
| 61 | POPAD | Restore registers |
| E9 0F 9E FF FF | JMP 00402FCC | Jump to unpacked code |

Table 5.9: Excerpt from Welchia's unpacking code

packers may only ever decrypt a small portion of the entire program at a time, decrypting the rest only as needed. This obviously has a performance penalty, and is more likely to be found in legitimate applications such as copy-protection than in malicious programs.

### Spawning multiple processes and disinclination to be terminated

In general, it is not normal for a user-mode program to spawn child processes. In Windows, system processes run as 'children' of the main System process (and Services run in turn as child processes under Services.exe). User-mode processes run either under explorer.exe or under CMD.EXE if started at the command line. Thus, a process other than those named above which has many 'children' may be suspicious.

This could potentially be used to detect the many malicious programs which take countermeasures against a user who terminates their main process. It is common for programs which display pop-up advertisements to do this, because they need to be visible to the user, yet not be easy for the user to terminate. If a pop-up advertisement window has appeared on the screen, it is not difficult to locate the process which displayed the window and kill it. However, most such programs have a second process whose job it is to restart the first process whenever it gets terminated – and finding the second process is often very difficult, because it only runs for a fraction of a second. In a similar way, malicious programs can 'fight back' against attempts to remove files or registry entries associated with them.

### Altering permissions and writing to system areas

Many types of malware copy themselves to system folders in order to hide among the many legitimate executables that are found there. The new executable will be given an innocuous-sounding name, so that even if the user is in the habit of looking in system folders she is still unlikely to notice.

When it first runs, the Asylum server copies itself to the Windows folder of the target computer, giving the new file the name 'wincmp32.exe' (though this can be changed via configuration). This name is clearly designed to mislead the user by appearing to be a system program.

It should be clear that the Asylum server will fail if the current user does not possess write access to the Windows folder. Many versions of Windows extant at the time Asylum was written (e.g. Win95/98/Me) still did not have a proper system of permissions, and allowed any user full write access to everything. Windows NT, 2000 and XP have stronger file permissions, but they are often effectively disabled by default (as in the case where users are permanently logged on as Administrator), so Asylum can still infect such systems. Other types of malware may use 'privilege escalation' exploits to get around this problem. A typical privilege escalation exploit is described in the article [Hückmann, 2006] which shows how to use the 'at' task scheduler to escalate privileges on some versions of Windows XP. Essentially, the problem occurs because the at command runs with system privileges and any processes it starts also have system privileges. Thus, it is possible to get access to Windows at an even lower level than an Administrator can. It has been reported that on some versions of Windows XP even the 'Guest' account (the lowest-privileged user) can use this vulnerability.

### Attempting to capture data from other programs

Some sorts of malware are designed to 'steal' data such as passwords from other programs. Windows has a number of (legitimate) mechanisms which make it possible for one program to access objects (such as windows) belonging to another program and capture the data they contain. These mechanisms are used legitimately by programs such as pop-up blockers (which need to find advertisement windows and close them) and programs for revealing 'asterisked' passwords in edit fields. Of course, such mechanisms can also be used by malware to steal passwords from web browsers and FTP clients. Malware may also hijack windows of other programs (usually web browsers), display their own content (such as a fake password entry page), and capture data in

that way. Possibly the cleverest example of data capture is illustrated by the E-Gold Trojan referred to in Chapter 3, which can actually siphon funds out of a target user's account during legitimate use of that account [LURHQ, 2004].

### Writing to specific Registry keys

The Asylum backdoor is capable of writing to various different parts of the Registry. The specific area it writes to is determined by the configuration process and by the version of Windows it is running on. The version of the Asylum server tested here added a key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Winlogon\Shell
```

having value 'explorer.exe wincmp32.exe'. This had the effect of making Windows execute the file 'wincmp32.exe' whenever the system was started. There are other registry areas commonly used by malware, including the keys 'Run' and 'RunServices' which are used by legitimate programs that need to run on startup. Asylum may also use these areas, depending on how the server was configured.

Earlier versions of Windows used configuration files ('WIN.INI' and 'SYSTEM.INI') instead of or concurrently with the Registry. Since Asylum was written in 2000 when such versions of Windows were still in use, it is also capable of writing to these files if they are present. Table 5.10 is an excerpt from the disassembled source code of the Asylum server, illustrating how the server writes to different registry keys depending on the value of preset configuration option variables.

The code given in Table 5.10 will now be explained. If the first call to the decision function does not return 1, the function designated 'Regwrite()' is called with the second argument being the address of the string 'Run', whereas if the second call to the decision function does not return 1, the Regwrite() function is called with the second argument being the address of the string 'RunServices'. This indicates how the configuration system is used to decide which registry keys are modified by the Asylum server. The Regwrite() function does the actual work of writing to the registry using the API functions RegOpenKeyEx() and RegSetValueEx().

| Machine Code | Assembly | Comment |
|---|---|---|
| 68 0C 32 40 00 | PUSH 0040320C | Address of decision variable |
| E8 53 0B 00 00 | CALL 00401CDC | Call decision routine |
| 83 F8 01 | CMP EAX,1 | Was the result 1? |
| 75 16 | JNE 004011A4 | If not, jump over next five instructions |
| 6A 01 | PUSH 1 | Arg of Regwrite(?) |
| 68 BA 31 40 00 | PUSH 004031BA | Arg of Regwrite() "SystemAdministration" |
| 68 A8 31 40 00 | PUSH 004031A8 | Arg of Regwrite() "Run" |
| 68 02 00 00 80 | PUSH 80000002 | Arg of Regwrite(?) |
| E8 C2 08 00 00 | CALL Regwrite() | |
| 68 0E 32 40 00 | PUSH 0040320E | Address of decision variable |
| E8 2E 0B 00 00 | CALL 00401CDC | Call decision routine |
| 83 F8 01 | CMP EAX,1 | Was the result 1? |
| 75 16 | JNE 004011C9 | If not, jump over next five instructions |
| 6A 01 | PUSH 1 | Arg of Regwrite(?) |
| 68 D4 31 40 00 | PUSH 004031D4 | Arg of Regwrite() "SystemAdministration" |
| 68 AD 31 40 00 | PUSH 004031AD | Arg of Regwrite() "RunServices" |
| 68 02 00 00 80 | PUSH 80000002 | Arg of Regwrite(?) |
| E8 9D 08 00 00 | CALL Regwrite() | |

Table 5.10: Registry Access Code from Backdoor.Win32.Asylum

### Suspicious resource usage

On one hand it might be thought that a successful malicious program should use as few system resources as possible. Whilst this is certainly a goal which might help malicious programs stay undetected for longer, in practice many malicious programs are resource-hogs (indeed, often the first thing that alerts a user to the possibility of a malware infection is that their computer or their network connection mysteriously slows down). There are three main reasons for this. The first and most trivial reason is that the malicious software might have explicitly been designed to hog resources – in effect, a denial of service attack against the local user.

The second reason is that a lot of malware is very badly written; either the malware authors are bad or inexperienced programmers, or they are more concerned with getting their malware out into the wild than with debugging it properly. Malware authors must necessarily remain anonymous, and there is little incentive for 'quality control'; furthermore, it has been known for new malware variants to be released containing bugs which were not present in the original version ([Szor, 2005] p354–355 cites some examples), which is generally due to deliberate mutation (e.g. a polymorphic or metamorphic virus, or a file infector which 'hitches a ride' on a worm) or to the 'script

kiddies' of the malware-writing community building on the work of more experienced authors. Sometimes bugs cause a particular piece of malware to be far more destructive than its author intended. The classic example here is the Morris Worm, which was covered in chapter 2.

The third reason could be considered deliberate strategy where specific types of malware are concerned. Intelligent malware authors know that their creations are inevitably going to be detected at some point. Thus, they might reason, it is best to make them do the maximum amount of work as soon as possible after infection, even if that carries a greater risk of detection. One type of malware which would benefit from this strategy would be a botnet for spamming. The longer each bot is on a user's computer the greater the risk of detection. Thus, it might be better to send several thousand spams immediately, even though this will almost certainly alert the user to a problem, since the chances are that by the time the user has taken steps to identify and remove the bot the majority of messages will have been sent (and of course, tying up the system makes it much harder for the user to do anything short of pulling the plug). The alternative would be to send only a small number of spams per day and risk having the bot caught by a routine virus scan before it has been able to send more than a few hundred messages. Similarly, a worm might be written to send out the maximum possible number of copies of itself, thus ensuring the greatest probability of passing on the infection. Obviously this "lightning strike" strategy would not be ideal for all types of malware, but could definitely benefit some.

According to many anti-malware databases, the Asylum server is known to cause resource problems, though none were encountered whilst running the Asylum server in the test environment. It is possible that different Asylum variants cause different problems.

#### List of Behaviours: A Summary

Of course, providing a list of behaviours is not the same as extracting real time information pertinent to the list. So it is necessary to consider what information can realistically be obtained and the methods that might be used to obtain it.

### 5.3.2 Behavioural Data Gathering Methods

As has been seen, gathering structural data are relatively easy: it is simply necessary to read each program file and write out data about it. Unfortunately, behavioural

monitoring is a much harder problem, as data must be collected in real time while the target process is running. There are three data-collection approaches that can be taken, each of which has its advantages and disadvantages.

1. Simulate processes 'offline'.
2. Run processes on an emulator.
3. Collect data from a live system.

### Simulation

Many forms of simulation applicable to process behaviour are possible, from theoretical models based on cellular automata through to 'virtual processors' which model program execution. Simulators have the advantage that as only simulated operations are being performed, malicious code can be simulated without harm. Their main disadvantage is that it is almost impossible to create a simulator complicated enough to accurately reflect real-world behaviour without it basically becoming an emulator (see below). The essential difference between simulators and emulators is that a simulator is essentially an abstract model of a system, often giving details of internal states etc., whereas an emulator aims to seamlessly reproduce the system. Simulators such as the Servile Software Decoder [Probert, 1994] were used in early virus research in order to study the operation of viruses without running them on a real processor.

### Emulation

Emulators and Virtualisers are essentially near-perfect simulators which usually also provide a realistic user interface clone for the target system. Whereas a simulator may only simulate processor operations an emulator will generally recreate memory, graphics and other aspects of the target system, sometimes to such a degree that casual examination is insufficient to determine whether a system is running natively or being emulated. The difference between an emulator and a virtualiser is that an emulator recreates the entire system from the ground up (including the processor) which allows software written for the emulated processor to be run on any other processor. Virtualisers, though, are usually designed to provide isolated access to the 'host' processor for a 'guest' operating system or program, and so can only run 'guests' written for the same processor.

One of the best-known commercial virtualisers is VMWare, which will emulate an Intel-based PC and run practically any PC operating system, including Microsoft Windows and GNU/Linux. Furthermore, the VMWare emulator will run *on* any of these systems. Thus, one can have Windows inside Windows, Windows inside Linux, Linux inside Windows or any other combination. Microsoft have also produced various virtualisation products including Virtual Server, drawing criticism from VMWare that they were limiting market choice [VMWare, 2007].[6] A large number of other virtualisation technologies and products exist, including Xen, an open-source virtual machine monitor originally developed at the University of Cambridge [Barham *et al.*, 2003]. Emulators and virtualisation environments are indispensable tools for human antivirus researchers. However, sufficiently advanced malware may refuse to execute in a virtual environment.[7]

#### Running Live Virus Code

This must of course be done on a protected and isolated computer or network to avoid the risk of passing on infections. However, if an isolated environment can be set up this offers the best and most practical means of analysis. Such an approach will be used here.

### 5.3.3 Two Means of Monitoring

There are effectively two approaches to monitoring activity on a system, which could be described as 'top-down' and 'bottom-up'. The top-down approach looks at the system activity as a whole, comparing memory and disk usage statistics, numbers of open files, and other metrics. Theoretically the presence of an unknown program ought to generate a 'ripple' of some sort in the overall system activity. The bottom-up approach, on the other hand, involves examining processes in memory as they run, and determining if they have changed or are acting suspiciously.

[6] But since VMWare is currently the market leader in virtualisation software, many commentators have taken their criticisms with a pinch of salt.

[7] This ought to be impossible in a pure emulated environment, but in practice depends on how the emulator/virtualiser is implemented.

### 5.3.4 Windows Performance Data

Most UNIX-like operating systems include a number of tools for enumerating running processes and gathering statistics about them (e.g. 'ps' and 'top'), whereas Windows has only the Task Manager. However, Windows keeps lists of performance data for every process which is executing. This data may be accessed in a number of ways, either through a special registry key (HKEY_PERFORMANCE_DATA) or via special API functions. The API functions will obtain the data for a process for which the application has a valid *process handle.* Whilst it is possible to access this data using the standard registry access API functions, this is an involved and complicated process.

#### The PSAPI Functions

However, Microsoft have provided two separate suites of functions which provide a simpler interface for accessing this data (EnumProcessModules() etc. in the PSAPI.DLL library and CreateToolhelp32Snapshot()/Process32First() in the 'TOOLHELP.DLL' library, though only the PSAPI functions will be considered for the purpose of this discussion). Thus, it is relatively easy to produce a background process which runs on a machine for a given period of time while the machine is in normal use, which can scan the process list and retrieve data for analysis. Such tools will give an idea of how processes are behaving, but only at a relatively high level. This is more akin to the top-down approach mentioned above. The PSAPI function EnumProcessModules() returns a list of ID values for each currently running process. Each process may then be opened using the standard API function OpenProcess(). If necessary, access may be gained to the memory mapped process file, and it may be examined in the same way as the program file above. However, since structural monitoring is not a concern here, functions such as GetProcessMemoryInfo() are more useful. This is a function which queries the performance data from the registry and returns a PROCESS_MEMORY_COUNTERS structure which contains a set of values concerned with the virtual memory usage of the process. The values returned are shown in Table 5.11.

Other information about a process such as the number of 'modules' it has mapped into its address space, the 'Owner' of the process, the number of threads it has, and so on, may also be obtained.

| Value | Type |
|---|---|
| Page Fault Count | number |
| Peak Working Set Size | bytes |
| Working Set Size | bytes |
| Peak Paged Pool Usage | bytes |
| Paged Pool Usage | bytes |
| Peak Non Paged Pool Usage | bytes |
| Non Paged Pool Usage | bytes |
| Page File Usage | bytes |
| Peak Page File Usage | bytes |

Table 5.11: Contents of a PROCESS_MEMORY_COUNTERS structure

### The Undocumented NtSystemQueryInformation() Function

Another very promising method involves the use of an undocumented[8] function in ntdll.dll called 'NtQuerySystemInformation'. This is the function which the Windows Task Manager uses to get its data on running processes. This function allows the obtaining of a list of all the running processes on a system, with details on each, including the process ID, the process name, the ID of the parent process, the number of threads and handles the process has, and the time the process has spent in user mode and kernel mode. Memory usage statistics for the process can also be obtained. Microsoft cautions against using NtQuerySystemInformation() because it is undocumented and may not be supported by future versions of Windows, though this is not a problem for the present purpose. It provides more information than the PSAPI or the ToolHelp functions, and moreover provides it in a single function call, whereas the previous methods generally require an API call per data item.

A list of most of the specific data provided by NtQuerySystemInformation() is given here. When called with an information class parameter of 'SystemProcessInformation', the function returns an array of structures of which not all the members are documented by Microsoft.

1. Process name
2. Process ID
3. ID of parent process

[8]Microsoft do actually document this function in MSDN, but advise against its use and do not give full details of the structures it returns, which others have had to reverse-engineer.

4. Handle count
5. Thread count
6. Virtual Memory statistics (pointer to a VM_COUNTERS structure giving same data as GetProcessMemoryInfo())
7. Creation time
8. Time spent in user mode
9. Time spent in kernel mode

A more detailed description of various fields useful for behavioural data collection is given in Chapter 7.

### 5.3.5 Lower-Level Methods

In order to spread in a Windows environment, a file-infecting virus must call API functions such as FindFirstFile() and FindNextFile() to discover new executable targets to infect, and must then call WriteFile() to infect them. If the monitoring tools can intercept or somehow observe this sequence of API calls coming from a program, it is pretty certain that it is exhibiting malicious behaviour, since very few 'normal' programs would have need to repeatedly search for executable files and write to them. Such observation is usually quite difficult to achieve for all API calls, and would require either off-line emulation or debugger-like attachment to each process. Off-line emulation (see below) is computationally expensive and does not amount to observing process behaviour 'in the wild'. Debugging, on the other hand, has its own special disadvantages, not least of which is that a debugger can never detach itself from a 'debuggee' without terminating it.

#### API Call Monitoring

Monitoring API calls is a little more difficult, but could certainly be done (for specific APIs) via mechanisms such as DLL injection. Windows provides functions that allow a programmer to force all processes running on a given computer to map a given DLL into their address spaces. A function in the DLL then has access to the program's address space. Getting the appropriate function called is sometimes difficult, but there

are various ways to achieve this, as documented in [Richter, 1995] and [Kuster, 2003]. Curiously, malware also uses these mechanisms, and one of the best ways to find out about them is to research so-called *user-land rootkits* – much of the information in this section was taken from a document which described such a rootkit [Kdm, 2005]. Once a DLL is mapped into the address space of a target process, API functions called by the DLL will refer to the target process, so any API function which allows a process to get information about itself can be called in order to get this information for the target process. It is possible to go even further and actually redirect API calls made by a target process, which is a method many userland rootkits use to hide files. API redirection is not only used for nefarious purposes, however – Microsoft have developed a method of using API redirection to implement 'instrumentation' of a program without needing to recompile its source code [Hunt and Brubacher, 1999].

#### Using a Device Driver for Detecting New Processes

A device driver can be used to trigger an event when a new process is started. It is thus possible to monitor every process. This technique is commonly used by personal firewall software such as ZoneAlarm. However, implementation of a device driver is a complex process and outside the scope of this PhD, and no suitable pre-written device driver was available for use.

### 5.3.6 Collecting Other Relevant Behavioural Data

Aside from performance data, there are a number of other things that it would be useful to know about a given process. This section details some of them.

#### Process Trees

The system performance data for a given process includes details of the 'parent' process, which is defined as the process which started it running. If the given process is a system process (a device driver or system service) its parent will generally be the main System process. Services are generally children of the process 'services.exe' which is in turn a child of the System process. User programs are generally children of 'explorer.exe' if the user starts them by clicking an icon. If invoked from the command line, processes become children of the command prompt process 'CMD.exe'. By doing some simple calculations it is possible to determine how many child processes have been invoked by

a given process. This could potentially be useful for classification, though there was insufficient time to devise a way to exploit it in this work.

#### Network Endpoints

Certain types of malware, such as bots, backdoors and worms, require the ability to initiate or receive connections over a network. Thus, in the monitoring process it would be helpful to be able to capture data on this. Luckily, Microsoft provide a set of functions allowing the TCP and UDP connection tables to be retrieved. These tables give details of which processes 'own' a given endpoint, allowing the number of endpoints per process per 'sample' to be retrieved.

### 5.3.7 Behavioural Monitoring: Conclusion

The final method chosen for the behavioural monitoring program used a loop which called NtSystemQueryInformation() a given number of times at specified time intervals. The results were fed into a special data structure which automatically collated together different 'samples' for each attribute of each process. A form of 'run-length encoding' was used to convert the list of samples for each attribute into a single number for classification purposes. For full details of this process and the exact data attributes used, see Chapter 7.

As always, there are constraints that should be noted. Firstly, the way in which the data are stored (a nest of linked list structures) assumes that the process ID is unique and does not get reused for another process during the period of monitoring. If this does happen, data for the new process will continue to be added to the record for the old process which used the same ID.

Another difference between the method of collecting structural and behavioural data are that, because structural data was collected from program files on disk, it was possible to collect all the data for each category of malware separately by organizing the malware files by category. However, since behavioural data must be collected from running processes, the data that ends up being collected will be mixed (depending on what malicious processes are running), and will have to be separated out. Thus, the pre-processing 'burden' is greater for behavioural data than for structural data. Furthermore, the behavioural data sets will be an order of magnitude smaller than the structural data sets for the simple reason that it is impractical to run tens of

thousands of malicious programs. Chapter 7 will discuss how these constraints may affect the results.

# Chapter 6

# Structural identification experiments

## 6.1 Introduction

This chapter is an investigation of whether or not, based on a library of known malware, using existing machine learning techniques, it is possible to automatically detect significant structural features and therefore to classify hitherto unseen programs as benign or malicious. Essentially, according to section 1.4.1, it is necessary to determine if *simple* attributes, not requiring extensive static analysis or other computation to obtain, can be used for classification. In order to test the hypotheses, experiments will be run classifying every category of malicious software against every other category of malicious software and against nonmalicious software.

## 6.2 Methodology

A large library of malicious software was obtained from an online source [VX, 2006]. The malicious software in the library was pre-classified into four categories: backdoors, Trojans, viruses and worms. A specially written data-collecting program was used to glean simple structural data, as described in Table 6.1, from the library. The data was output in a suitable format to be fed to the decision tree tools.

It was decided for reasons of practicality to restrict testing to Win32 PE (Portable Executable) format files only (detecting malicious scripts is a much more complex problem which would require different techniques). Thus, any code which was not in this format was ignored. This did not present a problem as the malware library contained sufficient samples of each category in the correct format.

Nonmalicious code samples for comparison were obtained later via a similar process (see below). The J4.8 WEKA decision tree classifier was used. In practice, the results produced by this classifier were mostly identical to those produced by C4.5,[1] but WEKA produces a more comprehensive set of output statistics than does C4.5. The other WEKA classifier was the Naive Bayes classifier. Figure 6.1 outlines the data-gathering process.

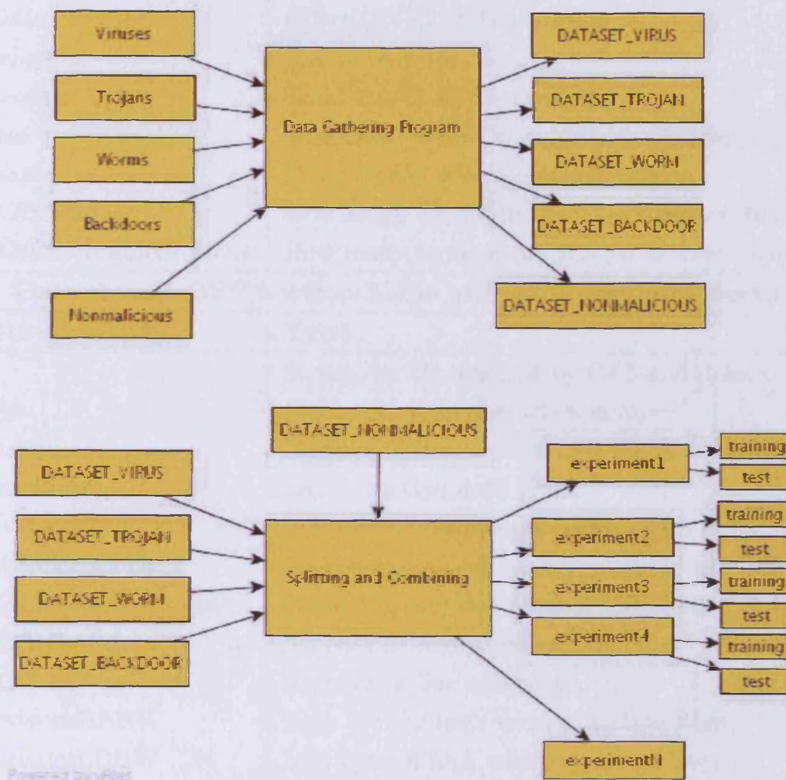



Figure 6.1: Data Gathering Process

### 6.2.1 The data-gathering program

This was a specially written program which would open each file, read in the headers and other information and log this to a file in a format appropriate for direct reading by the classification systems. Both C4.5 and WEKA use comma-separated value (CSV) format for input data, but class and attribute information is provided in different ways – for full details see Chapter 4.

[1]As they should be since the same algorithm is used. However, in practice the two systems sometimes produced slightly differing results when classifying identical datasets, possibly due to the way each system handles different number bases.

| Identification Data | |
|---|---|
| **Attribute** | **Type** |
| Name | String, for ID, ignored by C4.5 and deleted by WEKA |
| **Data from File Header** | |
| **Attribute** | **Type** |
| ImageBase | Memory location where loader maps program file |
| SizeOfCode | Total size of code section(s) |
| BaseOfCode | Offset (RVA) of code section |
| BaseOfData | Offset (RVA) of data section |
| SizeOfImage | Size of mapped file |
| SizeOfHeaders | Total size of all file headers |
| CheckSum | Sum of all WORDS in file plus file size |
| EntryPoint | Place (RVA) where execution starts |
| NumberOfSections | How many entries in the COFF section table? |
| NumberOfExecutableSections | How many sections are flagged as executable? |
| **Data from COFF Section Table of First Executable Section** | |
| **Attribute** | **Type** |
| Name | String, for ID, ignored by C4.5 and deleted for WEKA |
| VirtualSize | Final size of section when mapped |
| VirtualAddress | Address of first section byte when mapped (RVA) |
| SizeOfRawData | Size of section data in file |
| PointerToRawData | Pointer to first page of section data in file |
| PointerToRelocations | Pointer to any relocation entries in file |
| PointerToLineNumbers data | Pointer to any line number data in file |
| NumberOfRelocations | Number of relocation entries |
| NumberOfLineNumbers | Number of line numbers |
| CharacteristicsHBHW | High byte of high word of section flags |
| CharacteristicsLBHW | Low byte of high word of section flags |
| CharacteristicsHBLW | High byte of low word of section flags |
| CharacteristicsLBLW | Low byte of low word of section flags |
| **Calculated Data for File** | |
| **Attribute** | **Type** |
| NumberOfImportedFunctions | How many entries in the IAT? |

Table 6.1: Structural Attributes Used for Classification

Table 6.1 shows the structural data collected from the datasets. In keeping with the goal of simplicity, much of this can simply be read from the PE/COFF file headers, but some of it (such as the number of imported functions) must be calculated, though this is not an onerous calculation. To simplify data gathering, not all the fields of the file header were used, though it would be relatively simple to extend the process to include them. The term RVA means "relative virtual address" and usually refers to an address relative to the Image Base within a mapped executable file.

For a detailed description of the meanings of each of these fields, refer to [Microsoft, 2006]. Since WEKA does not allow any string data to be used (and has no 'ignore' attribute like C4.5), the two Name fields in the data were removed when WEKA was used, which was possible since they were being ignored by C4.5 anyway and were only there to aid identification of particular samples. Also, it should be noted that in initial tests the source datasets were purged of items which obviously referred to compressed (packed) executables (i.e. those with a first executable section called '.UPX0') in case the results were unduly influenced. However, after further consideration it was decided to abandon this practice, as even though it is possible for nonmalicious executables to be compressed or packed, it is significantly more common for malware to be so treated.

It is worth noting that different compilers, assemblers and other programs which create executable files may lay them out in different ways, and that this may affect some of the structural data. However, because malware authors are presumably equally as likely to use a specific compiler than other software authors, it is anticipated that such differences will occur with equal frequency in the malicious examples as in the nonmalicious examples, and can thus be considered as "background noise" which can be discounted. Certainly it is anticipated that the structural differences between malicious and nonmalicious software will be greater than the structural differences between two pieces of software produced using different compilers.

#### Collecting the nonmalicious samples

Nonmalicious samples were collected from a computer which had previously been scanned by a fully-updated signature scanner. Unfortunately, it is very common for duplicate copies of the same executable to be kept in different directories, so many duplicates tended to appear in the dataset, and these had to be purged out before it could be used.

### 6.2.2 The source datasets

Each source dataset was named as follows:

- DATASET-VIRUS: 1658 virus data samples from the malware library.
- DATASET-TROJAN: 1644 Trojan data samples from the malware library.
- DATASET-BACKDOOR: 1614 backdoor data samples from the malware library.

- DATASET-WORM: 592 worm data samples from the malware library.
- DATASET-NONMALICIOUS: 2062 program data samples selected from a scanned PC.

### 6.2.3 Creating experimental datasets

It was desired throughout to match an equivalent number of malicious and nonmalicious samples in each experimental dataset. This was done by determining the number of samples in each malicious source dataset and then randomly selecting an equivalent number of samples from DATASET-NONMALICIOUS. Ideally, the size of the experiment datasets should be divisible by two, if this was not the case a subset of the closest value divisible by two was selected.

The following process was used. In this description it will be assumed that two categories are being compared, denoted $A$ and $B$. Datasets associated with a category will be denoted $D[A]$ and $D[B]$. For random selection, a program was written using an implementation of the Mersenne Twister PRNG [Matsumoto and Nishimura, 1998].

### 6.2.4 Splitting into Training and Test sets

- Split $D[A]$ and $D[B]$ into equal parts $D_{Train}[A], D_{Test}[A]$ and $D_{Train}[B], D_{Test}[B]$.
- Combine opposite parts, creating $D_{Train}[A] \cup D_{Train}[B]$ (the training set) and $D_{Test}[A] \cup D_{Test}[B]$ (the test set).
- Add the appropriate data for C4.5 (a .names file) or WEKA (attribute information).

## 6.3 Results

For full results see Appendix A. Due to the volume of results produced only summaries will be given here.

Although all experiments were performed using C4.5 and WEKA, only the full outputs from WEKA are reported in the relevant tables. WEKA produces a larger range of output statistics than C4.5, one of which, the Kappa Statistic, requires some explanation.

### 6.3.1 The kappa statistic $\kappa$

The *kappa statistic*, also known as Cohen's kappa, [Cohen, 1960] can be used as an index which compares classification performance of the system against that which might be expected by chance. For two categories it is necessary to compute the proportion $p_c$ of instances which have been correctly classified, and the proportion $p_e$ which might be expected to be correctly classified by chance. These quantities are readily obtained from a confusion matrix. If the classifier performs perfectly then $p_c = 1$, so the maximum value of $p_c - p_e$ is $1 - p_e$. Therefore the kappa statistic is defined so as to normalise with respect to this maximum, viz.

$$\kappa = \frac{p_c - p_e}{1 - p_e} \tag{6.1}$$

Kappa values range from 1 (complete agreement) through 0 (agreement equivalent to chance) to -1 (complete disagreement). There is a simple generalization to more than two categories [Fleiss, 1981].

### 6.3.2 Baseline

It can be argued that since the uninfected sources for the infected files in the virus library are (presumably) not known, then it is likely to be very difficult to separate their characteristics from the characteristics of the viruses which are using them as 'hosts'. A counter to this is provided by cross-validating DATASET-NONMALICIOUS against itself: the dataset is divided in half and each half is assigned an arbitrary category. If the hypotheses hold, the classification system ought to perform no better than chance (50%) when classifying this data. The results in Table 6.2 illustrate this hypothesis rather convincingly.

### 6.3.3 Learning curves

Before presenting the experimental results in detail, in order to provide an overview, a comparison of the learning curves for discrimination between the categories Mal-NonMal and also between Virus-NonMal is given here.

In Figures 6.2 and 6.3 both training and the unseen test set size were 1264 and 1260 instances respectively. For example, for the Mal-NonMal experiment the training and

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correctly Classified Instances | 340 (53.7975%) | 316(50.0000%) |
| Incorrectly Classified Instances | 292 (46.2025%) | 316(50.0000%) |
| Kappa statistic | 0.0759 | 0.0000 |
| Mean absolute error | 0.4834 | 0.5015 |
| Root mean squared error | 0.4917 | 0.6232 |
| Relative absolute error | 96.6892% | 100.2963% |
| Root relative squared error | 98.3306% | 124.6420% |
| Total number of instances | 632 | 632 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correctly Classified Instances | 306 (48.4177%) | 306 (48.4177%) |
| Incorrectly Classified Instances | 326 (51.5823%) | 326 (51.5823%) |
| Kappa statistic | -0.0316 | -0.0316 |
| Mean absolute error | 0.5074 | 0.5090 |
| Root mean squared error | 0.5153 | 0.6289 |
| Relative absolute error | 101.4742% | 101.8001% |
| Root relative squared error | 103.0536% | 125.7817% |
| Total number of instances | 632 | 632 |

Table 6.2: Baseline

test sets were constructed as follows. First 1264 examples were randomly selected from the complete Malware set (Backdoor-Trojan-Virus-Worm), and then split this into two disjoint sets, each of 632 Malware examples. The same procedure was repeated for the Non-Malware set. Finally, these four sets were combined to produce training and test sets each containing 632 examples of Malware and NonMalware respectively. The

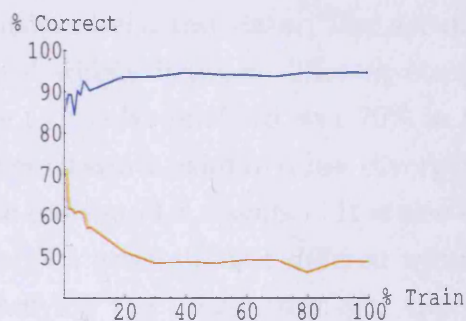



Figure 6.2: Learning curves for Mal-NonMal (Top: J48; Bottom: Naive Bayes). Experiment 5(2), section A.5.2.

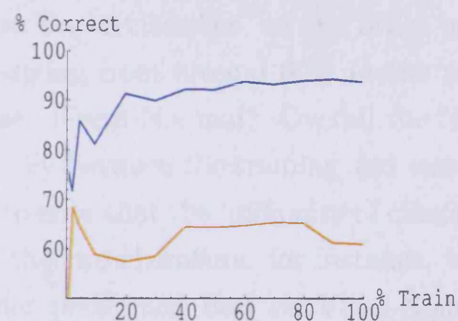



Figure 6.3: Learning curves for Virus-NonMal (Top: J48; Bottom: Naive Bayes). Experiment 1, section A.1.1.

two sets of size 1264 were then order-randomised. A similar process was used for the Virus-NonMal experiment.

In each figure the horizontal axis represents the percentage of the training set so far used, and the vertical axis represents the percentage of correctly categorised instances in the test set.

From these figures it is clear that the Bayesian classifier is having difficulty, whereas the decision-tree approach is performing well.[2] Whilst the J4.8 curves provide clear evidence of learning, the Naive-Bayesian classifier eventually performs no better than random with a final kappa of 0.1854. It looks as if the dependencies within the input attributes may be sufficient to effectively undermine the utility of the Naive-Bayes classifier. It is also interesting to note that J4.8 consistently manages to get discrimination at the 70%+ level on as few as 15 or so training instances.

### 6.3.4 Experiment One

In Experiment One, each malware dataset in turn was compared to the nonmalicious data set. The results showed that the J4.8 classifier was able to distinguish between the classes with a high degree of accuracy. The performance of the Naive Bayes classifier was much worse, though it still managed to correctly classify the majority of cases.

The following graph shows the percentage of cases correctly classified by each classifier in Experiment One.

It can be seen that the J4.8 classifier performed fairly consistently in each case, maintaining a very high degree of classification accuracy (exceeding 90%) even on the previously unseen test data. The accuracy of Naive Bayes classifier, on the other hand, varied widely between different comparisons, ranging from around 60% in the worst case (Virus-Nonmal) to over 70% in the best case (Worm-Nonmal). Overall the Naive Bayes classifier exhibited less divergence in accuracy between the training and test sets than did the J4.8 classifier. It is also interesting to note that the 'difficulty of classification' of a given dataset differed widely between the two classifiers: for instance, when classifying the unseen test set, the J4.8 classifier performed best on Virus-Nonmal, whereas the Naive Bayes classifier performed worst on the same dataset.

[2] Clearly, the performance of a classifier is substantially dependent on the *type* of data with which it is presented.

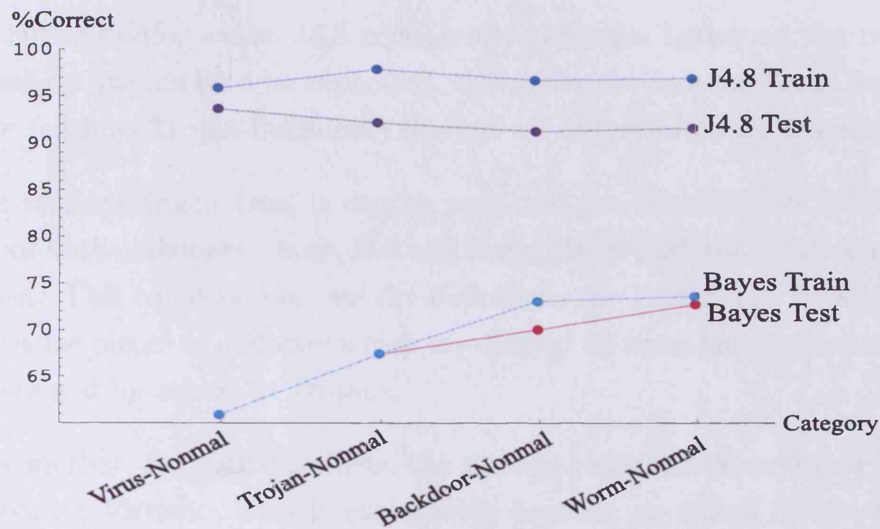



Figure 6.4: Structural Experiment One

### 6.3.5 Experiment Two

Experiment Two was a pairwise comparison of the malicious categories. Each possible pair of malicious categories was compared, so as there are 4 malicious categories, the total number of necessary comparisons is given by $\binom{4}{2} = 6$.

The following graph shows the percentage of cases correctly classified by each classifier in all six comparisons in Experiment Two.

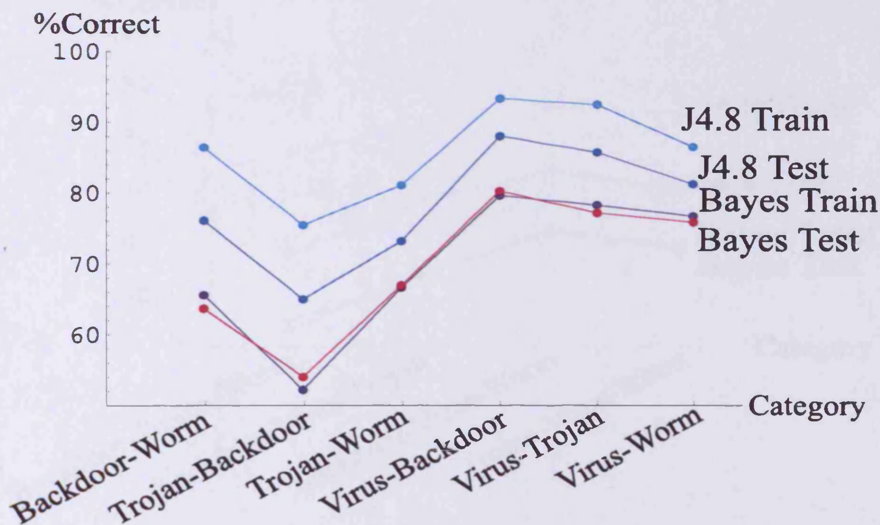



Figure 6.5: Structural Experiment Two

As in Experiment One, the J4.8 classifier is performing consistently better than the Naive Bayes classifier. But now there is much less consistency between different

datasets. Furthermore, whilst J4.8 consistently performs better on the training set than the test set (which is to be expected), this is not the case for Naive Bayes, where on occasion (such as Trojan-Backdoor) the test set outperforms the training set.

In contrast to Experiment One, it can be seen that the classification difficulty is the same here for both classifiers - both J4.8 and Naive Bayes perform worst on the Trojan-Backdoor set. This could be because the definitions for Trojan and Backdoor overlap - there are some pieces of malware which are defined by some anti-malware companies as Backdoors and by others as Trojans.

It also appears that, for both classifiers, the greatest classification accuracy is achieved when comparing Viruses. This is explainable because, as stated before, viruses are more likely to exhibit the sort of structural anomalies that the classifier will pick up.

### 6.3.6 Experiment Three

Experiment Three compares all possible triplets of malicious categories, the total number of necessary comparisons is given by $\begin{pmatrix} 4 \\ 3 \end{pmatrix} = 4$.

The following graph shows the percentage of cases correctly classified by each in Experiment Three.

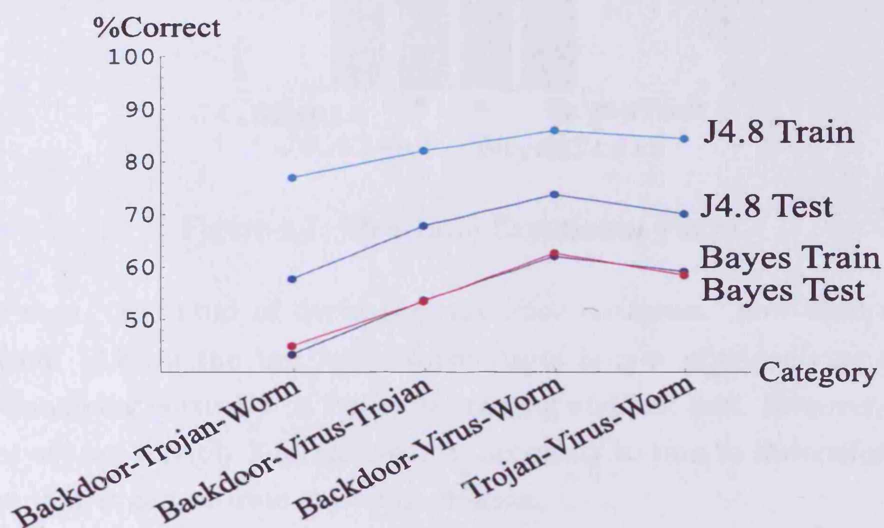



Figure 6.6: Structural Experiment Three

There appears to be a trend of decreasing accuracy - it has been falling since Experiment One. Here the worst results (for the Naive Bayes classifier on Backdoor-Trojan-

Worm) dip below 50% for the first time. The best that J4.8 manages on the test set is just over 70%.

There is also greater divergence between J4.8 training and test. In Experiments One and Two, J4.8 performed overall about 10% worse on the test set than on the training set. In this experiment, the divergence is closer to 20%. By contrast, the Naive Bayes classifier exhibits almost no divergence – the results for training and test sets are virtually identical.

### 6.3.7 Experiment Four

Experiment Four compared all four malware categories with each other. Thus, only one set of results was produced, which are given in Figure 6.7 below.

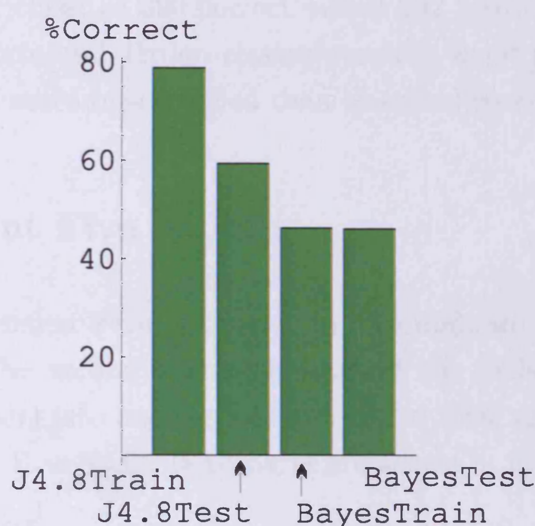



Figure 6.7: Structural Experiment Four

As can be seen, the trend of decreasing accuracy continues: now have under 60% accuracy from J4.8 on the test set. Naive Bayes is now misclassifying more cases than it is classifying correctly on both the training and test sets. However, since bare percentages are not terribly informative, it is necessary to turn to the confusion matrix in the hope that it can provide more information.

Table 6.3 shows the 'Confusion Matrix' produced by J4.8. This shows how many cases from each class were correctly classified and how many were misclassified as belonging to other classes. Each row corresponds to an actual malicious class, and the columns show how many were classified as belonging to each class.

| Actual/Classified As | Backdoor | Trojan | Virus | Worm |
|---|---|---|---|---|
| Backdoor | 390 | 145 | 25 | 72 |
| Trojan | 194 | 302 | 79 | 57 |
| Virus | 32 | 68 | 510 | 22 |
| Worm | 69 | 72 | 57 | 103 |

Table 6.3: Structural Experiment Four: J4.8 Confusion Matrix

For instance, it can be seen that that 390 cases whose actual classes was 'Backdoor' were correctly classified, but 145 further Backdoor cases were mistakenly classified as Trojans, 25 as Viruses and 72 as Worms. Similarly, in the case of Trojans, 194 were mistakenly classified as Backdoors. Of the four malware classes, the Virus class had the highest proportion of correct classification (510 cases correctly classified versus 122 incorrectly classified, amounting to 80% accuracy for the Virus class alone). The next best was the Backdoor class, at 390 correct versus 242 incorrectly classified cases, or 61% accuracy. The Worm and Trojan classes were the worst performers, both having a greater proportion of cases misclassified than classified correctly.

### 6.3.8 Experiment Five

The first part of Experiment Five compared all four malware categories plus the non-malicious category. The second part combined all the malicious categories (Virus, Worm, Trojan, Backdoor) into one big category called 'Mal' and compared it with the nonmalicious category. Results for both parts are shown in Figure 6.8.

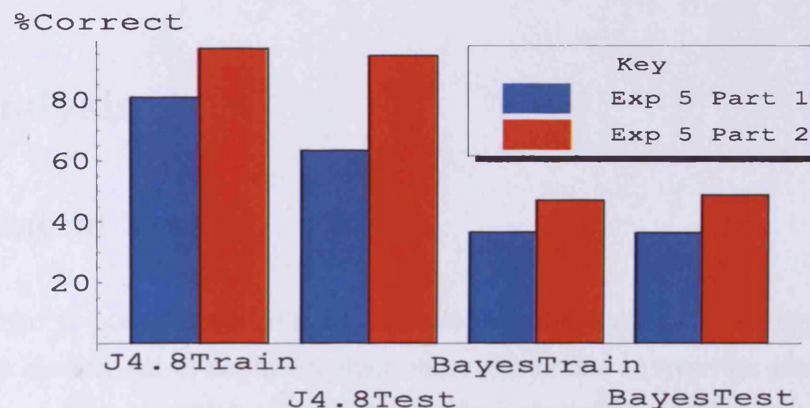



Figure 6.8: Structural Experiment Five

Considering the results for Part One, as might have been expected, the accuracy drops further. As in Experiment Four,the J4.8 confusion matrix, given in Table 6.4, will be

used to examine the individual accuracy rates for each class.

| Actual/Classified As | Backdoor | Trojan | Virus | Worm | Nonmal |
|---|---|---|---|---|---|
| Backdoor | 347 | 182 | 24 | 61 | 18 |
| Trojan | 172 | 326 | 67 | 46 | 21 |
| Virus | 26 | 66 | 485 | 27 | 28 |
| Worm | 62 | 100 | 55 | 75 | 9 |
| Nonmal | 29 | 27 | 8 | 8 | 560 |

Table 6.4: Structural Experiment Five (Part One): J4.8 Confusion Matrix

Comparing this confusion matrix (Table 6.4) with that of Experiment Four (Table 6.3) shows that adding the extra Nonmal class has had a significant impact on the individual classification accuracies of the already-existing classes. Simple calculation shows that the class with the greatest proportion of correctly-classified instances is Nonmal, with 89% correct. Next is Virus, with 77% correct. Backdoor and Trojan have 55% and 51% respectively. For Worms only 25% of cases were correctly classified, the majority apparently classified as Trojan. On the subject of false negatives, it is interesting to note that about 2% of Backdoors, 3% of Worms and Trojans, and 4% of Viruses were misclassified as Nonmal. Classes Trojan and Virus seemed to generate the most false positives (4%).

Looking at the Part Two results, it appears that on amalgamating all the malicious categories, the accuracy of J4.8 returns to levels not previously seen since Experiment One. Naive Bayes, on the other hand, is still unable to classify the majority of instances correctly.

## 6.4 Analysis

### 6.4.1 General Trends

A general trend is observable from the results: as the number of categories (classes) increases, the classification accuracy decreases. Thus the three-class classifiers of Experiment 3 were less accurate than the two-class classifiers of Experiments 1 and 2. This is in accordance with the so-called 'curse of dimensionality' (described in [Tan *et al.*, 2006]), which states that the classification accuracy of a classifier is inversely proportional to the number of classes.

In the baseline test, it was seen that J4.8 tended to outperform Naive Bayes – this trend continued in all experiments, with Naive Bayes on average making 20% fewer correct classifications than J4.8. J4.8 always managed to correctly classify the majority of cases, whereas Naive Bayes did not. However, the Naive Bayes classifier showed a much smaller difference in accuracy between training and test data than did J4.8, the accuracy of which tended to drop between 6% and 20% when classifying the test data. Naive Bayes, though it performed worse overall, showed a much smaller drop-off and sometimes was better able to classify the test data than the training data.

### 6.4.2 Specifics

Experiment 1 showed that all four classes of malware were easy to distinguish from the nonmalicious dataset, if taken individually. However, it was clear from Experiments 2 to 5 that distinguishing the four classes *from each other* was by no means as easy.

In all cases, the J4.8 classifier outperformed the Naive Bayes classifier, especially as the number of classes increased. For the largest datasets (Experiments 4 and 5 part 1) the Naive Bayes classifier actually misclassified the greater proportion of both the training and test data. It is also notable that, in accordance with the hypotheses (Section 1.4.1), the class of malware most likely to contain structural anomalies (viruses) is also the easiest to classify.

For certain datasets, it was possible to identify reasons for the significance of certain factors in the classification process. For example, experiments with the virus dataset indicated that samples with a lower ImageBase attribute were more likely to be viruses than nonviruses. The simplest explanation for this is that standard Windows application programs have a default ImageBase of 0x00400000. System programs, built-in Windows applications, and device drivers all have higher default ImageBase values, and these files are less likely to be infected by viruses because Windows has features which protect them from arbitrary modification. Thus, a sample of virus-infected programs will likely consist of a higher proportion of user programs than system ones. Section flags were also a significant factor in classification – which is likely to be because viruses which infect existing sections must often alter these. Results which show that smaller files are more likely to be malicious than larger ones are fairly self-evident, since any malicious program will be able to spread and evade detection more easily if it is small and compact (e.g. a Trojan travelling as an email attachment). However, many of the identified significant factors did not appear to have such explanations.

## 6.5 Conclusion

A practical system would benefit from the use of an *incremental induction* method of deducing a classifier, thus allowing continuous updating of classifiers with new malicious program data without the need to deduce a new classifier (although building a new classifier is not a particularly computationally expensive undertaking), see [Quinlan, 1993a] p106.

The baseline experiment of section 6.2 shows that the Mal/Nonmal classifiers constructed using randomly selected examples from the Nonmal data performed at a level indicative of a random classification, whereas the classifiers constructed using known identified examples from each category showed clear evidence of correct identification. In this way the hope expressed in the basic hypotheses of section 1.4.1 can reasonably be claimed to have been vindicated.

In practice normal users will not have a continually updated library of malware available for classifier/detector update. This process is in any event more easily carried out by existing Anti-Virus software, which could periodically download updated classifiers. Unlike signature based systems these malware-detectors would act as a first line of defense against hitherto *unseen* malware. The use of such systems could help bridge the vital time-gap, which exists at present, between the central detection of a new outbreak and the manual creation of a new signature file for download to subscribers computers. What the results of this chapter demonstrate is that the approach is perfectly feasible and offers a relatively high degree of protection.

Chapter 7

# Behavioural identification experiments

## 7.1 Introduction

This chapter discusses the experiments that were performed on behavioural data captured from running processes on an isolated computer. Continuing the discussion from Chapter 5, the experimental set-up, the methods used to capture the data, and the results obtained are described.

According to the hypothesis (Section 1.4.1), behavioural classification, if it works, ought to be equally good at classifying all types of malware (in contrast to structural classification, which, as previously shown, does in fact favour malware classes which have more structural anomalies). Thus, and also in order to be better able to make comparisons, the methodology used will be as similar as possible to that used in the structural experiments.

## 7.2 Methodology

Owing to the requirements for capturing data in real time, it was necessary to restrict testing to a much smaller set of programs than was used for the structural experiments. This was because, whereas hundreds or thousands of executable files can be structurally analysed in a reasonable amount of time, attempting to execute that number of files and analyse their behaviour is a much more intensive process which can easily slip beyond the bounds of what is reasonable (thus it might be said that this approach is less scalable). Furthermore, for these experiments at least it was necessary to choose malicious programs which were not known to interact with each other or to interfere

with other programs. This was to minimise the possibility of the monitoring software itself becoming infected, and to make the behavioural results as 'clean' as possible.

A set of specially selected malicious software examples from the malicious software library referred to in Chapter 6 were used. The test environment was a network-isolated computer running Windows XP. To prevent the possibility of infections leaving the computer, a one-way binary file policy was used as follows: the data gathering program was transferred to the isolated PC on a floppy disk, and was then deleted from the floppy disk. No binary files were allowed to remain on the floppy disk when transferred back to the work PC, and the floppy disk was scanned using an updated virus scanner at regular intervals to prevent any potential transfer of malware.

Difficulties encountered during the process of collecting the data included the fact that many examples from the malware library refused to run on the test environment. A further challenge was tracking down and removing traces of each piece of malware after data had been gathered. For this purpose a commercial virus scanner, previously tested to ensure that it recognised each piece of malware, was used.

## 7.3 The Data-Gathering Program

The behavioural data was collected by a specially-written program which used the undocumented Win32 API function NtQuerySystemInformation(), used by the Windows Task Manager. This function was called several times in succession to obtain a series of results for each process currently executing. As explained in Chapter 5, the choice to use this undocumented function was made because all the officially-documented methods of obtaining the same information required one function call per data item whereas NtSystemQueryInformation() returns a structure containing all the data at once, thus reducing overhead. As behavioural data are being captured in real time, this is important.

The program was also given the ability to start up a given process immediately prior to monitoring, and terminate it when monitoring had finished. Later, the data row corresponding to the process which was started up could be extracted from the output (which showed the data for all running processes).

### 7.3.1 Real-time Data

As a number of samples are being taken $n$ over a period of time $t$, it would be ideal if there was some way of encoding these into a single value, or definite set of values, for input to the classification system (which is much more convenient than trying to feed every value in, especially if $n$ is large). However, finding a suitable encoding process is not easy. Possibilities explored included runlength encoding, geometric (array) encoding and averaging groups of values. However, all except the averaging method proved to be either too difficult to implement satisfactorily or flawed in that they lost information. The chosen method was to average the $n$ samples into small groups of size $r$ where $n|r$. Setting $n = 1000$ and $r = 10$ meant that each group of 10 values was averaged, resulting in 100 averages.

The following table shows the data collected by the monitoring program.

| **Identification Data** | |
|---|---|
| **Attribute** | **Type** |
| ThreadCount | Number of threads owned by the process |
| HandleCount | Number of open handles (files, communication resources) |
| PeakVirtualSize | Maximum virtual size |
| VirtualSize | Size of address space |
| PageFaultCount | Number of page faults (pages swapped in from virtual memory) |
| PeakWorkingSetSize | Peak working set size (see below) |
| WorkingSetSize | Size of the process *working set*[1] |
| QuotaPeakPagedPoolUsage | Peak usage of the Paged Pool |
| QuotaPagedPoolUsage | Usage of the Paged pool (see [Russinovich and Solomon, 2005] p401) |
| QuotaPeakNonPagedPoolUsage | Peak usage of the Nonpaged Pool |
| QuotaNonPagedPoolUsage | Usage of the Nonpaged pool (see [Russinovich and Solomon, 2005] p401) |
| PageFileUsage | Usage of the page file |
| PeakPageFileUsage | Peak usage of the page file |
| NumTCPEndpoints | Number of TCP ports owned by the process |
| NumUDPEndpoints | Number of UDP ports owned by the process |
| ReadOperationCount | Number of read operations performed |
| WriteOperationCount | Number of write operations performed |
| OtherOperationCount | Number of other operations performed |
| ReadTransferCount | Number of bytes read |
| WriteTransferCount | Number of bytes written |

Table 7.1: Process Attributes Sampled Over Time

### 7.3.2 Choice of Malicious Examples

To prevent the possibility of cross-infection or infection of the monitoring software, the decision was made to exclude Viruses from the categories of malware examined in the experiments, thus leaving only the categories Backdoor, Trojan, and Worm. Owing

| Backdoor | Trojan | Worm |
|---|---|---|
| Asylum | ZoneKiller | Deborm |
| Blade | Starfield | Sasser |
| Brainspy | Gnome | SdBoter |
| AckCmd | Wintec | Welchia |
| AF | Zum | Lovesan |
| Daniel | Micron | Bymer |
| BO2K | Gnot.II | Donk |
| Oblivion.01 | ICQ2k | Poo |
| Breplibot | ICQPager | Qaz |
| Joiner | Mutapager | Zan |

Table 7.2: Malware used for behavioural data capture

to time constraints and the aforementioned limitation that only examples that would run on the test machine could be used, it was not feasible to employ random selection methods. Table 7.2 shows the malware that was used (names shortened for brevity). As there were too few cases to create separate training and test sets, it was decided to use 10-fold cross-validation instead (this procedure is explained in Chapter 4).

### 7.3.3 Final Datasets

1. Backdoor: a set of behavioural data for 10 backdoors.
2. Trojan: a set of behavioural data for 10 Trojans.
3. Worm: a set of behavioural data for 10 worms.
4. Nonmal: a set of data for 10 nonmalicious programs.

These datasets will be compared in the experiments that follow.

## 7.4 Results

Full tables of results are given in Appendix B. Here the most relevant results are presented in graph form.

### 7.4.1 Experiment One

Experiment One compared data from each malicious dataset against the Nonmal dataset. Figure 7.1 shows the results.

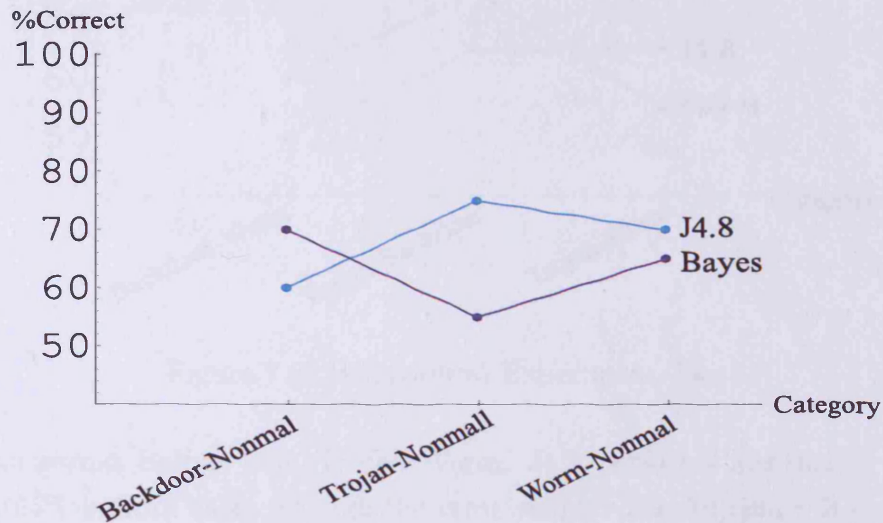



Figure 7.1: Behavioural Experiment One

It can be seen that for Backdoor-Nonmal, Naive Bayes is outperforming J4.8 significantly, correctly classifying 10% more instances. For Trojan-Nonmal J4.8 is achieving the highest proportion of correct classifications, though the percentage difference between the classifiers is about the same, though (here J4.8 achieves 20% more correct classifications than Naive Bayes whereas on Backdoor-Nonmal J4.8 achieved 20% *fewer* correct classifications). With Worm-Nonmal, the difference in percentage of correctly classified instances is much closer (5% as opposed to 20%). J4.8 still outperforms Naive Bayes, though less so than in Trojan-Nonmal, since J4.8 is performing about 5% worse in this comparison and Naive Bayes is performing about 10% better.

### 7.4.2 Experiment Two

Experiment Two compared each pair of malicious datasets. Figure 7.2 shows the results.

On Backdoor-Worm J4.8 performs poorly, achieving results identical to chance. Naive Bayes performs slightly better, but still worse than on any of the previously-seen datasets. Trojan-Backdoor is another case of Naive Bayes outperforming J4.8, though

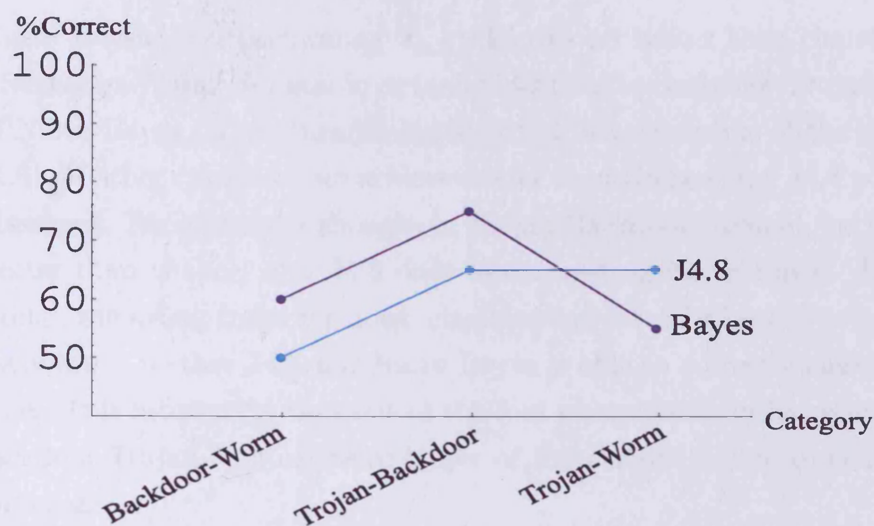



Figure 7.2: Behavioural Experiment Two

both do somewhat better. On Trojan-Worm, J4.8 performs identically to Trojan-Backdoor (65% in both cases, though the error values – see Appendix B – are slightly lower). Strangely, Naive Bayes performs 20% worse than it did on Trojan-Backdoor.

### 7.4.3 Experiment Three

Experiment Three compared each pair of malicious datasets plus the Nonmal dataset, and also all three malicious datasets together. The results are shown in Figure 7.3

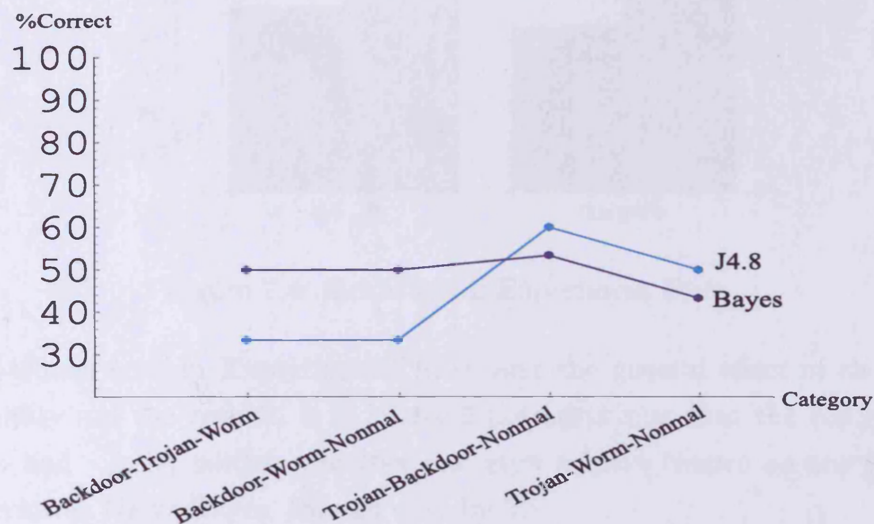



Figure 7.3: Behavioural Experiment Three

For Backdoor-Trojan-Worm, J4.8 manages to classify only 33% of cases correctly,

whereas Naive Bayes – outperforming it – performs no better than chance. Performance on Backdoor-Worm-Nonmal is virtually identical to Backdoor-Trojan-Worm (in the case of Naive Bayes the full results are identical, whereas some of the error values differ in J4.8). Neither classifier can achieve better than chance, and J4.8 achieves significantly less well. Paradoxically though, in Trojan-Backdoor-Nonmal, both classifiers perform better than chance, and J4.8 once again beating Naive Bayes. For Trojan-Worm-Nonmal, the losing trend resumes: classification rates for both types of classifier are extremely low – neither J4.8 nor Naive Bayes is able to correctly classify the majority of cases. It is interesting that out of the four comparisons in Experiment Three, only in Backdoor-Trojan-Nonmal were either of the classifiers able to achieve better results than chance.

### 7.4.4 Experiment Four

The first part of Experiment Four compared all four datasets. The second part combined all three malicious datasets into one class called "Mal", and compared it against the Nonmal dataset. Figure 7.4 shows the percentage of cases correctly classified in each part.

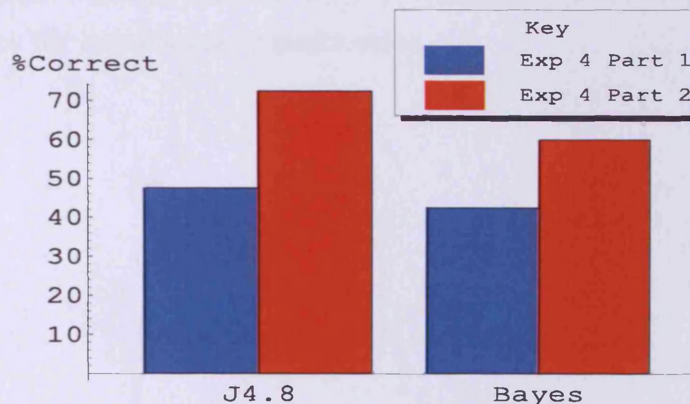



Figure 7.4: Behavioural Experiment Four

Given the trends seen in Experiment Three, and the general effect of the 'Curse of Dimensionality' on the results, it is perhaps not surprising that the results for Part One are so bad – here, neither classifier can even achieve chance accuracy. Even so, J4.8 outperforms Naive Bayes, though only by 5%.

In Part Two, on dispensing with the distinctions between different types of malware, and thus 'lifting' the Curse of Dimensionality, accuracy shoots back up again. However, referring to Appendix B, the Kappa statistics still do not look too healthy, even J4.8,

as it achieves 72.5% correct classification, can only manage 0.2903, little better than Naive Bayes' 0.2381. Thus agreement may still not be very good.

## 7.5 Conclusions

In general, it appears that it is feasible to distinguish each type of malware in turn from nonmalicious software based on the behavioural data used here. However, based on the behavioural characteristics that were easily collectable, accuracy is much less than when structural characteristics are used. Thus in distinction to the results of the previous chapter, it is not possible to claim with any great confidence that the hypotheses of section 1.4.1 have been clearly vindicated. Furthermore, when the classification system tries to distinguish *between* malware, or between multiple classes of malware and non-malware, accuracy nose-dives. This could be for many reasons. Firstly, it could be that sufficient data is not being captured to allow classification. Secondly, the encoding process used may be suboptimal. Thirdly, of course, it may be that it is genuinely much harder to classify based on behaviour than on structure, and possibly not feasible at all using only simple data. Given the constraints imposed by time and feasibility, it may be that the results in this chapter must be considered proof-of-concept rather than definitive evidence for behavioural classification.

# Chapter 8

# Conclusions and Future Work

## 8.1 Evaluation

It is clear from Chapter 6 that classifying based on *simple* structural attributes can give a high degree of accuracy, almost certainly comparable to the accuracy obtained by complex structural methods such as static analysis. However, the behavioural results given in Chapter 7 are by no means as good. Of course, behavioural classification is a much harder problem, and it is possible that truly good results cannot be obtained without the use of much lower-level and more comprehensive data capturing (for instance, that used by Microsoft in [Lee and Mody, 2006]). Certainly to collect behavioural data it is highly advantageous to be running in kernel mode, but writing the necessary drivers was beyond the scope and time allocation of this PhD project.

Accuracy notwithstanding, it is not the case that the methods employed in Chapter 6, if implemented as a complete system, could completely replace existing malware scanners – this was never the intention. However, it could provide a useful supplement to the existing systems which would have the ability to provide a degree of 'zero-day protection' from a new threat. It is arguable that a practical system would need to use some form of incremental induction, since unless the classification models have the capability to update themselves in response to new data (since it is possible that the structural or behavioural characteristics of new malware may change over time), the system would require regular classifier updates (just like a signature scanner) and could not be termed intelligent or flexible.

Any generalisations about the relative difficulty of discriminating malware using structural versus behavioural characteristics, must factor into consideration the practicality

of collecting the relevant characteristics in real time. To a significant degree what can be collected will be a function of the operating system. For behavioural charactistics the difficulty of generating low-level data collection software will be significantly determined by whether the OS is Open Source, or at least by the availability of information concerning the internal workings of the OS. At a meta-level it is fairly clear that in some sense malware must be determinable by behaviour, but is a far cry from this observation to a practical piece of software that can do this in realtime. Still, bearing these caveats in mind, overall, it seems that structural factors are not only easier to detect but result in higher classification accuracy than behavioural ones.

However, it is also possible that the high degree of structural difference between the non-malicious samples and the malware was caused by factors other than their maliciousness or lack thereof, though the fact that the structural baseline test (where a group of non-malicious samples was randomly assigned a category, divided in two and classified) performed poorly would seem to refute this. Certainly it was clear in both the structural and behavioural experiments that classifiers tended to pick up on general factors (e.g. malicious executables are usually much smaller than non-malicious ones) which were not by themselves necessarily good indicators of whether or not a given example was malicious or not.

## 8.2 The Original hypotheses and the final contribution

Were the hypotheses of section 1.4.1 vindicated?

In Chapter 6 it was shown that decision tree classifiers were indeed capable of discriminating malware on the basis of simple features elicited from executable files, whereas the performance of Naive Bayesian classifiers was less convincing. In addition, but with less accuracy, such classifiers were capable to some degree of classifying the *type* of malware.

As regards detection and classification on the basis of readily elicited *behavioural* features, partly as a result of the sheer time required to collect the necessary data, the results given in Chapter 7, although encouraging, were far less clear cut or convincing. More work would be needed to determine how realistic and/or how effective such an approach would be in practice.

In reality normal users will not have a continually updated library of malware available

for classifier/detector update. This process is in any event more easily carried out by existing anti-malware software, which could periodically download updated classifiers.

Unlike signature based systems the malware-detectors proposed herein would act as a first line of defense against hitherto *unseen* malware. The use of such systems could help bridge the vital time-gap, which exists at present, between the central detection of a new outbreak and the manual creation of a new signature file for download to subscribers' computers.

- The principal contribution of this thesis is to demonstrate that the decision-tree classifier approach, using easily collected structural data, is perfectly feasible and offers a relatively high degree of protection.

## 8.3 The PhD: A retrospective review

Looking back over the 3 years of this PhD, there are many things which could have been done differently or better. It would have been ideal to have obtained the library of malware examples earlier on, and to have set up an entire isolated network (see below) instead of just one machine. This would have provided a better environment for testing the behaviour of many types of malware which propagate across networks. It would also have been nice to explore the possibility of combining structural and behavioural detection. Some of these topics are covered in the Future Work section below.

At first, it was envisaged that this PhD would result in the development of a complete malware detection system. As the PhD progressed it soon became clear that such a goal was far beyond the scope of a 3-year PhD project.

The general impression received of the malware research field is that it is extremely 'cliquey", being dominated by commercial interests who are loath to provide information to outsiders. Attempts to contact researchers working in the field received polite but discouraging replies suggesting, in effect, that rather than "meddling in things we didn't understand" malware research should be left to the professionals. It is telling that, with a few exceptions such as Peter Szor's book [Szor, 2005], the only way to obtain accurate technical information about malware was to rely on 'underground' materials produced by groups of malware authors, who seem to exchange information freely with the anti-malware researchers, but who have less of a problem with sharing it with the wider community. There are of course good reasons for not providing just

anyone with potentially dangerous information, but the malware field is secretive even compared with other fields of computer security research, including some where the sensitivity of the information is perhaps greater.

## 8.4 Future Work

As there was a hard three-year deadline in place for this PhD, there was much that could have been done but which had to be put aside owing to insufficient time. This section lists some of these, as well as other ideas that were rejected owing to difficulty or which proved to be blind alleys.

### Other possible approaches for behavioural classification

There are some alternative paradigms for behavioural feature detection and classification which may well be worth investigating. A couple are mentioned here.

One possibility is *FSM induction.* Whilst malware in general need by no means belong to the restricted class of Finite State Machines, the fact remains that the input/output (I/O) behaviour of most malware is rather trivial and might in practical terms easily fall into the category of a FSM. If such I/O behaviour could be collected for a process, and an FSM constructed by induction on the I/O strings then it might be possible to identify and classify the malware by the FSM. Following the original paper of [Biermann and Feldman, 1972] numerous FSM induction algorithms have been proposed – so this approach, if it proved practical to collect such I/O data, would be based on a well understood methodology.

Another, more speculative, alternative would be to look at the *resource usage and allocation* within the machine (or network) as a whole. This gestalt provides an insight into the overall 'health' of the machine. Such monitoring could act as a 'tripwire' to sound the alarm when a change from 'healthy' to 'unhealthy' was detected.

### Authentication, or Identification of Friend or Foe

Assuming that the anomaly detection system has identified a program which is behaving abnormally. What should happen then? Should the offending program be deleted or terminated? The danger here is that a genuine program may trigger a 'false positive' in the detection system. One way of avoiding this is to use program authentication.

Program authentication involves a method by which a program can be examined and identified as possessing the right 'credentials'. Ideal authentication methods should be resistant to such tricks as stealing credentials from a valid program, replaying the credentials of a valid program, or infecting or subverting a valid program without changing its credentials. Unfortunately, most practical authentication methods fall short of these requirements.

Fairly basic authentication already exists in many operating systems, one example being the use of checksums. Under Windows,[1] program file headers contain a checksum field, where the linker or program which created the file can store a checksum value for the file. When the program runs, the program loader in the operating system recalculates its checksum and compares the new value with the one in the file header. If the checksums do not match, the loader can in theory refuse to run the program. In versions of Windows prior to Windows XP, checksums are generally set to zero and not checked at all, except for DLLs and critical system files [Pietrek, 1994 updated 2002]. Later versions of Windows are much more stringent, and insist that all critical files have an intact cryptographic signature.

Checksums were originally designed to detect accidental corruption, and for this they are perfectly adequate. They are not an adequate defence against viruses, however, as viruses are capable of calculating valid checksums for infected files. A better method involves generating checksums for all programs at installation using a cryptographic checksum algorithm such as MD5 or SHA, then storing them in a secure location accessible only to the authentication system. Programs can then be authenticated by recalculating the checksum and comparing it with the stored value. Systems to do this are available, but they all depend on the security of the stored checksum list. If a virus or Malicious Program can get write access to the list, it can change credentials at leisure. Thus, authentication methods have become more advanced. For instance, current versions of Windows support policies which restrict which executable files users can run, based on four factors. Software can be restricted by cryptographic hash, by certificate, by path or by internet zone.[2] However, like all authentication-alone systems, this relies on a system administrator imposing a central policy on all users. An even more advanced authentication procedure can be conceived, in two parts. The first part would take the form of a program responsible for patching all authorised executable files at installation with a small function which computes a response to a challenge.

[1]The Portable Executable format has a checksum field. The UNIX ELF format does not appear to have one.

[2]A guide is found at http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/rstrplcy.mspx – accessed 22/05/2007

The function will be exported, and the process can then be identified as authorised on calling its authentication function and receiving the correct response. The response will be based on checksums in such a way that if a process has been patched or infected by a virus, its response will no longer be valid. The functions used must be computationally cheap, but very difficult for an unauthorised process or third party to duplicate or forge.

For the second part, another program, which can be termed the 'verifier', is needed. The verifier sends a 'challenge' to the process as described above, and checks the response it receives. If the response is correct the process is allowed to 'pass' whereas if it is incorrect further action may be taken. This component may be implemented as a single program, or as a system of autonomous verification agents which roam the system or network at random eradicating unauthorised processes.

### 8.4.1 Anomaly Detection

The data mining/machine learning techniques used in this thesis concentrated on classification of programs into various categories of malware. An alternative method would be to use anomaly detection to detect which programs are most 'anomalous', on the assumption that anomalous programs are more likely to be malicious. This is a technique currently used for detecting fraudulent credit card transactions and network intrusions [Tan *et al.*, 2006]. Anomaly detection is likely to be an easier problem than classification, but in practice this technique might well be limited by the scale of 'anomalousness' used. Anomaly detection is likely to be most useful as a pre-screening technique. Either a classification method or an authentication method (see below) could then be applied as the second stage.

### 8.4.2 Combining Anomaly Detection and Authentication

Anomaly detection on its own has limited use due to the fact that false positives can never be entirely eliminated. Authentication on its own would require some strategy as to when to authenticate – and too much authentication could result in resource wastage. The ideal combination of anomaly detection and authentication would work by finding anomalies, then authenticating the programs detected by the anomaly detector. Assuming the authentication process is reasonably difficult to foil, this would allow the system to be reasonably sure that a program that is behaving anomalously and fails authentication is rogue and should be terminated (though see below). If

a program behaves anomalously but passes authentication, the response might vary depending on the needs of the system administrator.

If a process that is behaving suspiciously has been challenged by the verifier and has not given the correct response, one possibility would be to kill and delete the process. However, to guard against destroying a legitimate process which has produced a 'false positive' it might be safer to 'freeze' it and prompt the user for some action. If the user or administrator, for example, has installed a new program and forgotten to authorise it with the authentication engine, they may choose to let the program run. However, if the program is genuinely unauthorised it can be terminated.

### 8.4.3 A Network for Malware Research

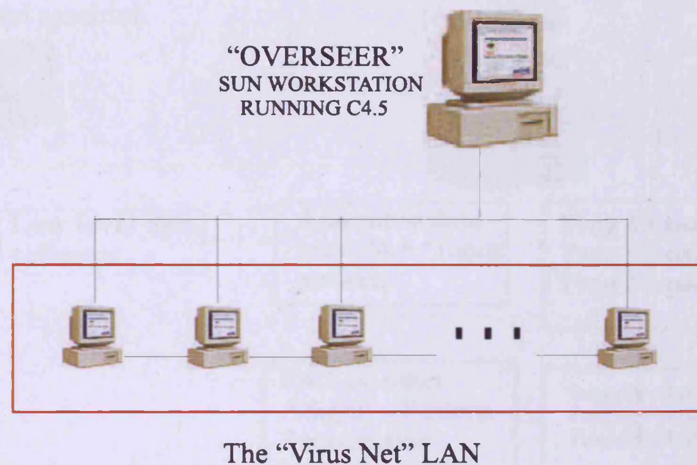



Figure 8.1: Example Network for Malware Research

As mentioned above, all the malware tests in this project were run on a single computer. This is not an ideal environment for testing certain types of malware which require a network to propagate. Furthermore, protecting a network is a different challenge from protecting a single computer. This section is a speculative design for a research network which could be used for the further development of the ideas developed in this PhD. Such a network is shown in Figure 8.1.

The research network would consist of an isolated network of Intel-compatible computers running the Microsoft Windows operating system. This architecture–operating system combination has been chosen because the vast majority of viruses are written for it. The network computers may be physical computers or virtual machines on VMWare or Microsoft Virtual Server – the advantage of using virtual machines is that

it is possible to create several machines from one image, thus ensuring that all the machines are identical.

One or more monitoring computers, preferably of a different architecture and running a different operating system to avoid the possibility of infection, will also be connected to the network. These will run network traffic analysis programs to allow all the network traffic to be logged. Each Windows PC will also run a variety of monitoring tools, and will submit results over the network to the monitoring machines. These monitoring tools will continually monitor the memory usage, disk usage and other pertinent statistics about each machine, together with a profile of each program running on them. The monitoring tools will constantly feed data back to a central location (for example, the host used by the system administrator). This host will run the analysis programs and can alert the administrator of potential threats.

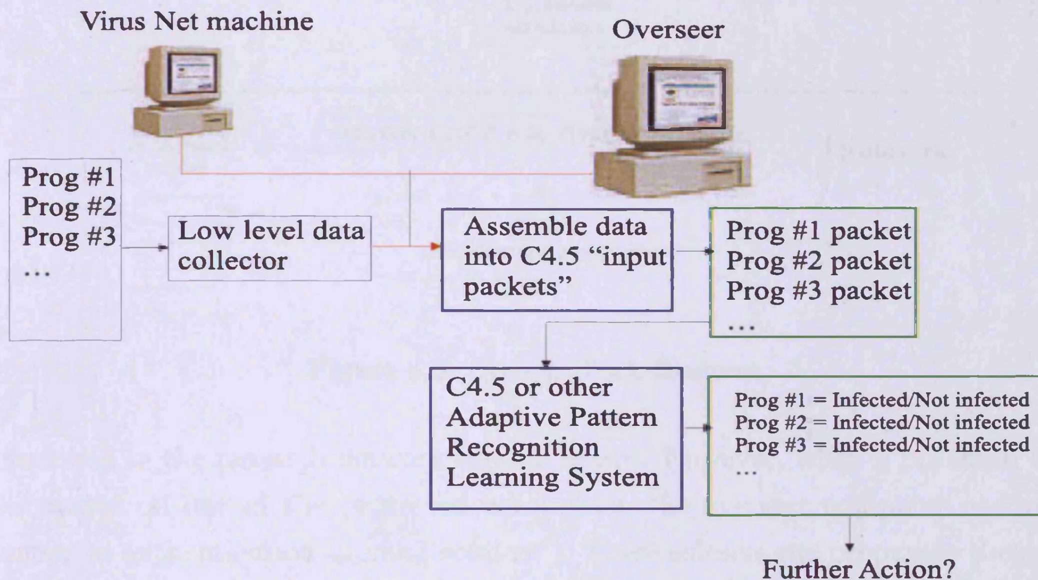



Figure 8.2: Data collection and detection of infection. Here any appropriate adaptive pattern recognition program could be used. (Process boxes blue, Data boxes green.)

Figure 8.2 shows how data flows around the research network.

### 8.4.4 A Complete Protection System

The following diagram (Figure 8.3) shows one way in which a complete protection system could be developed by combining the research network from the previous section with a 'Friend-or-Foe' identification scheme.

The scheme shown in Figure 8.3 will now be described. The monitoring process works

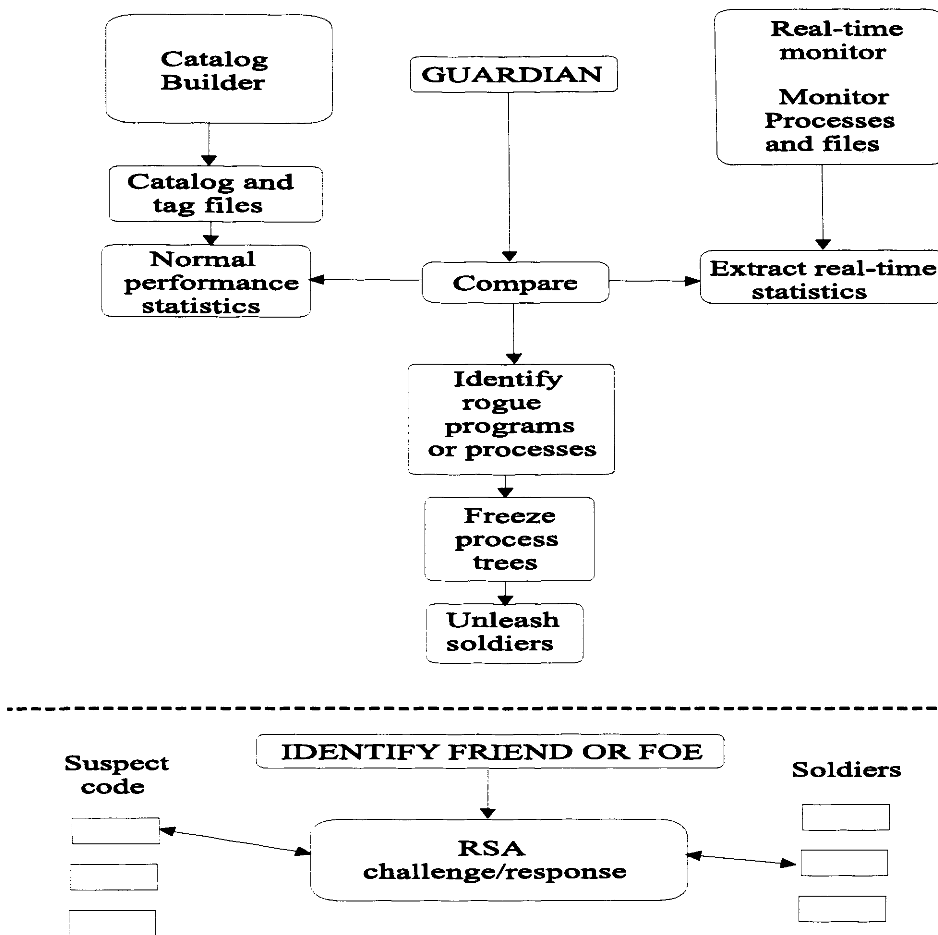



Figure 8.3: System Block Diagram

as described in the research network section above. However, when a potential threat is discovered on one of the protected computers, the overseer computer can release autonomous authentication agents ('soldiers'). These soldiers can propagate themselves over the network and will be programmed with information about the potential threat process. When a soldier encounters an instance of the potential enemy process on a protected network computer, it will issue a challenge to it. On receiving a correct response, the soldier will terminate itself and send signals to other soldiers and to the overseer that the process is not a threat. If an incorrect response is received, the soldier will terminate the enemy process. Depending on the overall strategy, it could then be programmed to issue 'terminate on sight' orders to other soldiers, meaning that the others could then dispense with the challenge and simply terminate any instance of the process they encounter.

The advantages of a system of this nature are several. Firstly, the Friend-or-Foe identification process minimises the chance of a benign process being targeted due to a false positive result from the detection scheme. Secondly, the agent-based design of the

'soldiers' means that the overseer or master monitoring program can swiftly respond to a potential threat discovered anywhere on the protected network, and can be sure of eradicating it even if it has spread to other computers. However, there would be difficulties in implementing a practical scheme of this nature. Firstly, it would be necessary to use as foolproof an authentication scheme for the Friend-or-Foe as possible, to ensure that malicious software cannot simply fake or steal authentication signatures.[3] This would almost certainly imply that every binary on every machine on the protected network would need to be individually granted a signature at the installation stage, though this may not be too onerous given that many corporate networks are based on master image files anyway. Secondly, the monitoring software and the software for supporting the soldier agents would add a significant amount of overhead to the network. But if these difficulties could be overcome, such a system would almost certainly be useful.

## 8.5 Final Thoughts

This PhD has been an interesting and challenging project in many ways. It has required combining information gathered from 'respectable' academic journals with that from underground 'zines', and to cite the work of both highly-regarded academics and notorious cyber-criminals.

Certainly at the beginning three years ago it was almost impossible to foresee what would "come out at the end" - whether the PhD would result in a revolutionary new malware protection system or no results of any value. In the circumstances, it has been demonstrated that classification of malware by structure using an automated classification system is feasible, and that classification by behaviour is at least possible. It has been demonstrated that decision tree classifiers tend to perform better on structural data than probabilistic classifiers such as Naive Bayes. On the broader question of whether this information will be of benefit to the fight against malware, the jury is still out. It is arguable that this particular war, like many others, will never be won; whatever innovations are made by one side will soon be countered by the other side. Sadly, in the digital as well as the physical world, human nature dictates that for every weakness there will always be someone ready to exploit it. Likewise, when faced with threats digital or physical, there is the temptation to retreat into a locked-down, restricted sandbox world, hoping that Big Brothers such as Trusted Computing, Microsoft, or Symantec will protect us, and accepting any restrictions they may place on

[3] Not considering here the question of whether or not such an authentication scheme is even feasible.

knowledge or freedom. For many people such short-term thinking may work, but in the long term it is unlikely to work so well.

> "Those who would give up Essential Liberty to purchase a little Temporary Safety, deserve neither Liberty nor Safety."
> – Benjamin Franklin (attr.),
> "An Historical Review of the Constitution and Government of Pennsylvania." (1759)

# Appendix A

# Detailed Structural Results

Given here are the detailed results from each experiment.

# A.1 Experiment One: detecting specific categories of malware

This first set of experiments compared malware of each class with nonmalicious data.

## A.1.1 Virus-NonMal

This experiment compared 1264 virus data samples with 1264 nonmalicious data samples, split equally into groups of 632 and divided into training and test sets using the procedure given above.

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1212 (95.8861%) | 770 (60.9177%) |
| Incorrect | 52 (4.1139%) | 494 (39.0823%) |
| Kappa statistic | 0.9177 | 0.2184 |
| Mean absolute error | 0.0743 | 0.3842 |
| Root mean squared error | 0.1923 | 0.551 |
| Relative absolute error | 14.8653% | 76.8483% |
| Root relative squared error | 38.4507% | 110.2052% |
| Total number of instances | 1264 | 1264 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1184 (93.6709%) | 770 (60.9177%) |
| Incorrect | 80 (6.3291%) | 494 (39.0823%) |
| Kappa statistic | 0.8734 | 0.2184 |
| Mean absolute error | 0.0961 | 0.3809 |
| Root mean squared error | 0.241 | 0.5504 |
| Relative absolute error | 19.2143% | 76.1808% |
| Root relative squared error | 48.1922% | 110.086% |
| Total number of instances | 1264 | 1264 |

Table A.1: Results from WEKA Classifiers Applied to Virus-NonMal

From these results it is seen that the J4.8 classifier maintains a high level of classification accuracy on both the training and test sets (95% and 94% respectively). The results for the Bayes classifier are paradoxical, since the numbers are mostly identical for the training and test sets.

### A.1.2 Trojan-NonMal

This experiment compared 1264 Trojan data samples and 1264 nonmalicious data samples.

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 1237 (97.8639%) | 853 (67.4842%) |
| Incorrect | 27 (2.1361%) | 411 (32.5158%) |
| Kappa statistic | 0.9573 | 0.3497 |
| Mean absolute error | 0.0394 | 0.3335 |
| Root mean squared error | 0.1404 | 0.5437 |
| Relative absolute error | 7.8855% | 66.7006% |
| Root relative squared error | 28.0811% | 108.7461% |
| Total number of instances | 1264 | 1264 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 1165 (92.1677%) | 853 (67.4842%) |
| Incorrect | 99 (7.8323%) | 411 (32.5158%) |
| Kappa statistic | 0.8434 | 0.3497 |
| Mean absolute error | 0.0914 | 0.3331 |
| Root mean squared error | 0.2696 | 0.5435 |
| Relative absolute error | 18.2762% | 66.6155% |
| Root relative squared error | 53.9183% | 108.7046% |
| Total number of instances | 1264 | 1264 |

Table A.2: Results from WEKA Classifiers Applied to Trojan-NonMal

Again, the J4.8 classifier maintains high accuracy, though slightly lower now than for the previous data set. The Bayes classifier again seems to give an identical performance classifying training and test data, though the error figures differ very slightly.

### A.1.3 Backdoor-NonMal

This experiment compared 1264 backdoor data samples and 1264 nonmalicious data samples.

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1222 (96.6772%) | 886 (70.0949%) |
| Incorrect | 42 (3.3228%) | 378 (29.9051%) |
| Kappa statistic | 0.9335 | 0.4019 |
| Mean absolute error | 0.0581 | 0.2913 |
| Root mean squared error | 0.1695 | 0.4934 |
| Relative absolute error | 11.6272% | 58.2599% |
| Root relative squared error | 33.9011% | 98.6762% |
| Total number of instances | 1264 | 1264 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1153 (91.2184%) | 924 (73.1013%) |
| Incorrect | 111 (8.7816%) | 340 (26.8987%) |
| Kappa statistic | 0.8244 | 0.462 |
| Mean absolute error | 0.1078 | 0.2644 |
| Root mean squared error | 0.2797 | 0.4699 |
| Relative absolute error | 21.569% | 52.8793% |
| Root relative squared error | 55.9366% | 93.971% |
| Total number of instances | 1264 | 1264 |

Table A.3: Results from WEKA Classifiers Applied to Backdoor-NonMal

The J4.8 classifier is still performing well, though the accuracy is decreasing compared to the previous two data sets. Oddly, the Bayes classifier does a better job of classifying the test data than the training data, and it is still not performing anywhere near as well as the J4.8 classifier.

### A.1.4 Worm-NonMal

Unfortunately only a relatively small number of worm data samples were available in the source dataset. The experiment therefore compared 602 worm data samples and 602 nonmalicious data samples, with training and test sets containing 301 of each.

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 583 (96.8439%) | 438 (72.7575%) |
| Incorrect | 19 (3.1561%) | 164 (27.2425%) |
| Kappa statistic | 0.9369 | 0.4551 |
| Mean absolute error | 0.0569 | 0.2651 |
| Root mean squared error | 0.1687 | 0.5036 |
| Relative absolute error | 11.3857% | 53.0191% |
| Root relative squared error | 33.7427% | 100.7189% |
| Total number of instances | 602 | 602 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 551 (91.5282%) | 443 (73.588%) |
| Incorrect | 51 (8.4718%) | 159 (26.412%) |
| Kappa statistic | 0.8306 | 0.4718 |
| Mean absolute error | 0.1042 | 0.261 |
| Root mean squared error | 0.278 | 0.504 |
| Relative absolute error | 20.8387% | 52.2051% |
| Root relative squared error | 55.5904% | 100.8038% |
| Total number of instances | 602 | 602 |

Table A.4: Results from WEKA Classifiers Applied to Worm-NonMal

Again, it is notable that the Bayes classifier performed better on the test set than the training set. However, whereas in classifying the Backdoor-Nonmal dataset the proportion of items correctly classified was about 3% higher for the test set than the training set, here it is only about 1% higher. This could be because the data set is smaller.

## A.2 Experiment Two: pairwise malware comparisons

This experiment compared every combination of two malicious classes, resulting in six sets of results:

| 1. Backdoor-Worm | 4. Virus-Backdoor |
|---|---|
| 2. Trojan-Backdoor | 5. Virus-Trojan |
| 3. Trojan-Worm | 6. Virus-Worm |

### A.2.1 Backdoor-Worm

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 806 (86.388%) | 612 (65.5949%) |
| Incorrect | 127 (13.612%) | 321 (34.4051%) |
| Kappa statistic | 0.6839 | 0.3455 |
| Mean absolute error | 0.2147 | 0.3475 |
| Root mean squared error | 0.3267 | 0.5298 |
| Relative absolute error | 49.1164% | 79.4738% |
| Root relative squared error | 69.8818% | 113.3384% |
| Total number of instances | 933 | 933 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 710 (76.0986%) | 594 (63.6656%) |
| Incorrect | 223 (23.9014%) | 339 (36.3344%) |
| Kappa statistic | 0.463 | 0.3206 |
| Mean absolute error | 0.3012 | 0.3589 |
| Root mean squared error | 0.4308 | 0.5429 |
| Relative absolute error | 68.8887% | 82.0956% |
| Root relative squared error | 92.1451% | 116.1383% |
| Total number of instances | 933 | 933 |

Table A.5: Results from WEKA Classifiers Applied to Backdoor-Worm

Both types of classifier are able to classify the majority of cases correctly, though there is an intriguing drop-off in accuracy of the J4.8 classifier - for the test data the proportion of cases classified correctly falls by about 10%, whereas for the Naive Bayes

classifier it only falls by about 2%. The greatest difference between the correctly-classified proportion of training and test data for J4.8 in Experiment One was around 5%.

### A.2.2 Trojan-Backdoor

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 953 (75.4553%) | 659 (52.1774%) |
| Incorrect | 310 (24.5447%) | 604 (47.8226%) |
| Kappa statistic | 0.5091 | 0.0442 |
| Mean absolute error | 0.3403 | 0.4741 |
| Root mean squared error | 0.412 | 0.678 |
| Relative absolute error | 68.0598% | 94.8117% |
| Root relative squared error | 82.4045% | 131.5521% |
| Total number of instances | 1263 | 1263 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 821 (64.9525%) | 683 (54.0348%) |
| Incorrect | 443 (35.0475%) | 581 (45.9652%) |
| Kappa statistic | 0.2991 | 0.0807 |
| Mean absolute error | 0.4104 | 0.4598 |
| Root mean squared error | 0.491 | 0.645 |
| Relative absolute error | 82.0827% | 91.9554% |
| Root relative squared error | 98.2054% | 128.9992% |
| Total number of instances | 1264 | 1264 |

Table A.6: Results from WEKA Classifiers Applied to Trojan-Backdoor

The trend of J4.8 outperforming Naive Bayes continues, as does the 10% drop-off between the training and test data for J4.8. Here also is another example of the Naive Bayes classifier achieving greater success with the test data than the training data (as was seen in several cases in Experiment One).

### A.2.3 Trojan-Worm

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 756 (81.1159%) | 621 (66.6309%) |
| Incorrect | 176 (18.8841%) | 311 (33.3691%) |
| Kappa statistic | 0.5394 | 0.173 |
| Mean absolute error | 0.2762 | 0.359 |
| Root mean squared error | 0.3712 | 0.4821 |
| Relative absolute error | 63.1442% | 82.0766% |
| Root relative squared error | 79.3794% | 103.0925% |
| Total number of instances | 932 | 932 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 683 (73.2047%) | 625 (66.9882%) |
| Incorrect | 250 (26.7953%) | 308 (33.0118%) |
| Kappa statistic | 0.3508 | 0.1867 |
| Mean absolute error | 0.3423 | 0.3569 |
| Root mean squared error | 0.4485 | 0.4812 |
| Relative absolute error | 78.2809% | 81.6127% |
| Root relative squared error | 95.933% | 102.927% |
| Total number of instances | 933 | 933 |

Table A.7: Results from WEKA Classifiers Applied to Trojan-Worm

Here, as in the previous results, a bug in the dataset-generation has resulted in the test set containing one more case than the training set, though owing to the number of cases it is unlikely this has affected the results. All previously noted trends continue, though the increase in accuracy for the Naive Bayes test set is much smaller.

### A.2.4 Virus-Backdoor

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1179 (93.3492%) | 1006 (79.6516%) |
| Incorrect | 87 (6.6508%) | 257 (20.3484%) |
| Kappa statistic | 0.867 | 0.5931 |
| Mean absolute error | 0.1184 | 0.2029 |
| Root mean squared error | 0.2429 | 0.4256 |
| Relative absolute error | 23.6708% | 40.5735% |
| Root relative squared error | 48.5706% | 85.1287% |
| Total number of instances | 1263 | 1263 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1113 (88.0538%) | 1015 (80.3006%) |
| Incorrect | 151 (11.9462%) | 249 (19.9664%) |
| Kappa statistic | 0.7611 | 0.606 |
| Mean absolute error | 0.1624 | 0.1979 |
| Root mean squared error | 0.3197 | 0.4216 |
| Relative absolute error | 32.4723% | 39.578% |
| Root relative squared error | 63.9394% | 84.3253% |
| Total number of instances | 1264 | 1264 |

Table A.8: Results from WEKA Classifiers Applied to Virus-Backdoor

Again, noted trends continue, though the drop in accuracy between the J4.8 training and test sets is only around 5% here, and the increase in Naive Bayes accuracy is also very small (less than 1%).

### A.2.5 Virus-Trojan

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1167 (92.4723%) | 988 (78.2884%) |
| Incorrect | 95 (7.5277%) | 274 (21.7116%) |
| Kappa statistic | 0.8494 | 0.5658 |
| Mean absolute error | 0.1285 | 0.2462 |
| Root mean squared error | 0.2528 | 0.4215 |
| Relative absolute error | 25.6994% | 49.235% |
| Root relative squared error | 50.5593% | 84.3066% |
| Total number of instances | 1262 | 1262 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1084 (85.7595%) | 976 (77.2152%) |
| Incorrect | 180 (14.2405%) | 288 (22.7848%) |
| Kappa statistic | 0.7152 | 0.5443 |
| Mean absolute error | 0.1923 | 0.2516 |
| Root mean squared error | 0.3521 | 0.4283 |
| Relative absolute error | 38.4598% | 50.3131% |
| Root relative squared error | 70.4225% | 85.6524% |
| Total number of instances | 1264 | 1264 |

Table A.9: Results from WEKA Classifiers Applied to Virus-Trojan

It seems that the virus data set is more distinctive than the others, as all the comparisons involving it seen so far in Experiments One and Two have produced consistently more correctly-classified instances in both J4.8 and Naive Bayes. In this case there is a 6% drop in the proportion of correctly-classified instances for J4.8 and a much lesser drop for Naive Bayes, which conforms to previous observations.

### A.2.6 Virus-Worm

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 806 (86.4807%) | 715 (76.7167%) |
| Incorrect | 126 (13.5193%) | 217 (23.2833%) |
| Kappa statistic | 0.6892 | 0.4073 |
| Mean absolute error | 0.2102 | 0.2329 |
| Root mean squared error | 0.3239 | 0.468 |
| Relative absolute error | 48.0568% | 53.2458% |
| Root relative squared error | 69.2568% | 100.0869% |
| Total number of instances | 932 | 932 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 758 (81.2433%) | 708 (75.8842%) |
| Incorrect | 175 (18.7567%) | 225 (24.1158%) |
| Kappa statistic | 0.5705 | 0.3857 |
| Mean absolute error | 0.2553 | 0.2438 |
| Root mean squared error | 0.3857 | 0.4818 |
| Relative absolute error | 58.3881% | 55.7545% |
| Root relative squared error | 82.5054% | 103.054% |
| Total number of instances | 933 | 933 |

Table A.10: Results from WEKA Classifiers Applied to Virus-Worm

The observations which applied to the previous set of results also apply here - it does seem that, for this experiment at least, the comparisons involving the virus dataset show less of a decrease in accuracy between classifying the training and test data than the comparisons involving the other datasets only.

## A.3 Experiment Three: three-way comparisons of malware

This experiment compared every combination of three malicious classes, resulting in four sets of results:

| 1. Backdoor-Trojan-Worm | 3. Backdoor-Virus-Worm |
|---|---|
| 2. Backdoor-Virus-Trojan | 4. Trojan-Virus-Worm |

### A.3.1 Backdoor-Trojan-Worm

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1204 (76.9821%) | 678 (43.3504%) |
| Incorrect | 360 (23.0179%) | 886 (56.6496%) |
| Kappa statistic | 0.6335 | 0.093 |
| Mean absolute error | 0.2234 | 0.3869 |
| Root mean squared error | 0.3333 | 0.5611 |
| Relative absolute error | 52.6096% | 91.116% |
| Root relative squared error | 72.3449% | 121.7678% |
| Total number of instances | 1564 | 1564 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 903 (57.6997%) | 704 (44.984%) |
| Incorrect | 662 (42.3003%) | 861 (55.016%) |
| Kappa statistic | 0.3297 | 0.1248 |
| Mean absolute error | 0.3278 | 0.3817 |
| Root mean squared error | 0.4642 | 0.5536 |
| Relative absolute error | 77.1951% | 89.903% |
| Root relative squared error | 100.7513% | 120.1532% |
| Total number of instances | 1565 | 1565 |

Table A.11: Results from WEKA Classifiers Applied to Backdoor-Trojan-Worm

Here is particularly bad performance from J4.8 on the test data - the proportion of correctly-classified instances decreases by nearly 20%. The performance of Naive Bayes is even worse, with the greater proportion of cases being incorrectly classified in both the training and test sets, though paradoxically there is a slight improvement (around 1%) in the result for the test set!

### A.3.2 Backdoor-Virus-Trojan

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 1554 (82.0486%) | 1015 (53.5903%) |
| Incorrect | 340 (17.9514%) | 879 (46.4097%) |
| Kappa statistic | 0.7307 | 0.304 |
| Mean absolute error | 0.1707 | 0.3294 |
| Root mean squared error | 0.2916 | 0.5165 |
| Relative absolute error | 38.4053% | 74.1057% |
| Root relative squared error | 61.8562% | 109.5685% |
| Total number of instances | 1894 | 1894 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 1285 (67.7743%) | 1014 (53.481%) |
| Incorrect | 611 (32.2257%) | 882 (46.519%) |
| Kappa statistic | 0.5166 | 0.3022 |
| Mean absolute error | 0.2506 | 0.3289 |
| Root mean squared error | 0.4055 | 0.5148 |
| Relative absolute error | 56.389% | 74.0032% |
| Root relative squared error | 86.013% | 109.2065% |
| Total number of instances | 1896 | 1896 |

Table A.12: Results from WEKA Classifiers Applied to Backdoor-Virus-Trojan

A slight improvement for J4.8 here with only around 14% decrease in the proportion of correctly classified instances, but still far worse performance than seen in the previous two experiments. Naive Bayes managed to correctly classify the majority of instances in both the training and test sets - the only variation being about 0.1%.

### A.3.3 Backdoor-Virus-Worm

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1344 (85.9335%) | 970 (62.0205%) |
| Incorrect | 220 (14.0665%) | 594 (37.9795%) |
| Kappa statistic | 0.7752 | 0.3967 |
| Mean absolute error | 0.1531 | 0.2501 |
| Root mean squared error | 0.2748 | 0.4614 |
| Relative absolute error | 36.0447% | 58.9014% |
| Root relative squared error | 59.6465% | 100.1459% |
| Total number of instances | 1564 | 1564 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1154 (73.7852%) | 980 (62.6598%) |
| Incorrect | 410 (26.2148%) | 584 (37.3402%) |
| Kappa statistic | 0.586 | 0.4092 |
| Mean absolute error | 0.2209 | 0.2494 |
| Root mean squared error | 0.379 | 0.4612 |
| Relative absolute error | 52.021% | 58.7294% |
| Root relative squared error | 82.2579% | 100.0915% |
| Total number of instances | 1564 | 1564 |

Table A.13: Results from WEKA Classifiers Applied to Backdoor-Virus-Worm

J4.8 improves again here, with 12% difference, and Naive Bayes somehow manages to achieve a 0.6% improvement on the test set (representing exactly ten more cases correctly classified).

### A.3.4 Trojan-Virus-Worm

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1319 (84.389%) | 924 (59.1171%) |
| Incorrect | 244 (15.611%) | 639 (40.8829%) |
| Kappa statistic | 0.7489 | 0.3213 |
| Mean absolute error | 0.1693 | 0.2795 |
| Root mean squared error | 0.2895 | 0.4653 |
| Relative absolute error | 39.8594% | 65.8072% |
| Root relative squared error | 62.829% | 100.9856% |
| Total number of instances | 1563 | 1563 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1097 (70.0958%) | 915 (58.4665%) |
| Incorrect | 468 (29.9042%) | 650 (41.5335%) |
| Kappa statistic | 0.5199 | 0.3108 |
| Mean absolute error | 0.2494 | 0.2823 |
| Root mean squared error | 0.4014 | 0.4678 |
| Relative absolute error | 58.7228% | 66.4727% |
| Root relative squared error | 87.1138% | 101.5367% |
| Total number of instances | 1565 | 1565 |

Table A.14: Results from WEKA Classifiers Applied to Trojan-Virus-Worm

Here J4.8 gets about 14% difference again, and Naive Bayes around 1%. As a general rule it seems that Naive Bayes performs about 20% worse than J4.8 in all experiments, though there is far less difference between performance on training and test sets for Naive Bayes than for J4.8.

## A.4 Experiment Four: four-way comparison of malware

This experiment compared all four malicious classes

### A.4.1 Backdoor-Trojan-Virus-Worm

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 1730 (78.8155%) | 1017 (46.3326%) |
| Incorrect | 465 (21.1845%) | 1178 (53.6674%) |
| Kappa statistic | 0.7089 | 0.252 |
| Mean absolute error | 0.1579 | 0.2832 |
| Root mean squared error | 0.2794 | 0.479 |
| Relative absolute error | 43.0822% | 77.2659% |
| Root relative squared error | 65.2653% | 111.8597% |
| Total number of instances | 2195 | 2195 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 1305 (59.3992%) | 1012 (46.0628%) |
| Incorrect | 892 (40.6008%) | 1185 (53.9372%) |
| Kappa statistic | 0.4436 | 0.2487 |
| Mean absolute error | 0.2376 | 0.2839 |
| Root mean squared error | 0.3924 | 0.479 |
| Relative absolute error | 64.826% | 77.45% |
| Root relative squared error | 91.6665% | 111.9046% |
| Total number of instances | 2197 | 2197 |

Table A.15: Results from WEKA Classifiers Applied to Backdoor-Trojan-Virus-Worm

Strangely, the introduction of the fourth category has not continued the decrease in performance (owing to the "curse of dimensionality"- it is still around 20% for J4.8. Naive Bayes misclassifies the greatest proportion of cases in both the training and test sets, with a decrease in accuracy of about 0.3% between them.

## A.5 Experiment Five: non-malware versus malware

In Part 1, all four malicious classes plus the nonmalicious class were compared (Backdoor-Trojan-Virus-Worm-NonMal). In Part 2, all four malicious class names were replaced with a single "Malicious" class and compared with the Nonmalicious dataset.

### A.5.1 Part 1: Backdoor-Trojan-Virus-Worm-NonMal

**Training Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 2284 (80.7924%) | 1033 (36.5405%) |
| Incorrect | 543 (19.2076%) | 1794 (63.4595%) |
| Kappa statistic | 0.7553 | 0.1852 |
| Mean absolute error | 0.1115 | 0.2684 |
| Root mean squared error | 0.236 | 0.4547 |
| Relative absolute error | 35.3143% | 85.0303% |
| Root relative squared error | 59.4065% | 114.4698% |
| Total number of instances | 2827 | 2827 |

**Test Data**

| | **J4.8** | **Naive Bayes** |
|---|---|---|
| Correct | 1793 (63.3793%) | 1030 (36.4086%) |
| Incorrect | 1036 (36.6207%) | 1799 (63.5914%) |
| Kappa statistic | 0.5338 | 0.1836 |
| Mean absolute error | 0.1677 | 0.268 |
| Root mean squared error | 0.3353 | 0.4556 |
| Relative absolute error | 53.1437% | 84.923% |
| Root relative squared error | 84.3961% | 114.6757% |
| Total number of instances | 2829 | 2829 |

Table A.16: Results from WEKA Classifiers Applied to Experiment 5 (part 1)

Although the overall accuracy has decreased as the number of categories has increased since Experiment One (owing to the "curse of dimensionality"), the maximum difference between the training and test sets for J4.8 has yet to exceed 20% (it being around 18% here). The Naive Bayes classifier is now hopelessly misclassifying 63% of cases in both the training and test sets.

## A.5.2 Part 2: Mal-NonMal

**Training Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 2742 (96.9933%) | 1330 (47.0463%) |
| Incorrect | 85 (3.0067%) | 1497 (52.9537%) |
| Kappa statistic | 0.9112 | 0.1679 |
| Mean absolute error | 0.0524 | 0.5287 |
| Root mean squared error | 0.1617 | 0.7259 |
| Relative absolute error | 15.1002% | 152.2548% |
| Root relative squared error | 38.805% | 174.2268% |
| Total number of instances | 2827 | 2827 |

**Test Data**

| | J4.8 | Naive Bayes |
|---|---|---|
| Correct | 2677 (94.6271%) | 1380 (48.7805%) |
| Incorrect | 152 (5.3729%) | 1449 (51.2195%) |
| Kappa statistic | 0.8407 | 0.1854 |
| Mean absolute error | 0.073 | 0.5109 |
| Root mean squared error | 0.2187 | 0.7132 |
| Relative absolute error | 21.0345% | 147.1465% |
| Root relative squared error | 52.5173% | 171.2266% |
| Total number of instances | 2829 | 2829 |

Table A.17: Results from WEKA Classifiers Applied to Experiment 5 (part 2)

As predicted by the "curse of dimensionality", once the number of categories is again reduced to two, overall accuracy shoots up to greater than 90% for J4.8. The drop-off between the training and test sets is now only around 2%. However, the Naive Bayes classifier is still hopelessly off the mark, the reduction of the number of categories not preventing it from misclassifying over 50% of cases.

# A.6 Summary

**Virus-NonMal**

| Result | Accuracy | TruePositive | FalsePositive | TrueNegative | FalseNegative | Precision |
|---|---|---|---|---|---|---|
| C4.5 | 0.94 | 0.94 | 0.06 | 0.94 | 0.06 | 0.94 |
| C4.5rules | 0.92 | 0.95 | 0.11 | 0.89 | 0.05 | 0.9 |
| J4.8 | 0.94 | 0.94 | 0.06 | 0.94 | 0.06 | 0.94 |
| NaiveBayes | 0.7 | 0.99 | 0.58 | 0.42 | 0.01 | 0.63 |

| Result | Accuracy | TruePositive | FalsePositive | TrueNegative | FalseNegative | Precision |
|---|---|---|---|---|---|---|
| C4.5 | 0.94 | 0.93 | 0.04 | 0.96 | 0.07 | 0.96 |
| C4.5rules | 0.93 | 0.91 | 0.04 | 0.96 | 0.09 | 0.96 |
| J4.8 | 0.94 | 0.93 | 0.04 | 0.96 | 0.07 | 0.96 |
| NaiveBayes | 0.64 | 0.99 | 0.71 | 0.29 | 0.01 | 0.58 |

**Backdoor-NonMal**

| Result | Accuracy | TruePositive | FalsePositive | TrueNegative | FalseNegative | Precision |
|---|---|---|---|---|---|---|
| C4.5 | 0.94 | 0.92 | 0.05 | 0.95 | 0.08 | 0.95 |
| C4.5rules | 0.94 | 0.91 | 0.04 | 0.96 | 0.09 | 0.96 |
| J4.8 | 0.94 | 0.92 | 0.05 | 0.95 | 0.08 | 0.95 |
| NaiveBayes | 0.72 | 0.99 | 0.55 | 0.45 | 0.01 | 0.64 |

**Worm-NonMal**

| Result | Accuracy | TruePositive | FalsePositive | TrueNegative | FalseNegative | Precision |
|---|---|---|---|---|---|---|
| C4.5 | 0.9 | 0.92 | 0.11 | 0.89 | 0.08 | 0.89 |
| C4.5rules | 0.92 | 0.95 | 0.1 | 0.9 | 0.05 | 0.91 |
| J4.8 | 0.9 | 0.92 | 0.11 | 0.89 | 0.08 | 0.89 |
| NaiveBayes | 0.73 | 0.99 | 0.53 | 0.47 | 0.01 | 0.65 |

Table A.18: Summary: Classifier Accuracy on Test Data

# Appendix B

# Detailed Behavioural Results

Given here are the detailed results from each experiment described in Chapter 7. Unlike the structural experiments described in Chapter 6 and Appendix A, separate training and test sets were not used. Instead, 10 fold cross-validation was used (see Chapter 4 for details).

Stratified cross-validation in WEKA produces a different set of output statistics which have not been encountered before. Definitions will be given for them here.

- **TP Rate** is the true positive rate, or the proportion of examples which were classified as class C, among all examples which are really in C.
- **Recall** is the same as TP Rate, but is given separately here because WEKA outputs it separately.
- **FP Rate** is the false positive rate, or proportion of examples which were classified as belonging to other classes than C but which in reality belong to C.
- **Precision** is the proportion of items which really belong to C out of all the items classified as belonging to C,
- **F-Measure** is a way of measuring precision and recall in one, calculated according to the equation given below.

$$F = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{B.1}$$

# B.1 Experiment One

## B.1.1 Backdoor-Nonmal

Table B.1 gives the results from running the J4.8 and Naive Bayes classifiers on Backdoor-Nonmal.

| Backdoor-Nonmal | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 12 (60.0000%) | 14 (70.0000%) |
| Incorrectly Classified Instances | 8 (40.0000%) | 6 (30.0000%) |
| Kappa statistic | 0.2000 | 0.4000 |
| Mean absolute error | 0.4000 | 0.3000 |
| Root mean squared error | 0.5852 | 0.5477 |
| Relative absolute error | 80.0000% | 60.0000% |
| Root relative squared error | 117.0466% | 109.5445% |
| Total number of instances | 20 | 20 |

Table B.1: Backdoor-Nonmal

It can be seen that Naive Bayes is outperforming J4.8 significantly here, correctly classifying 10% more instances. The Kappa Statistic also indicates better agreement for Naive Bayes than for J4.8, though in neither case is the agreement outstanding.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.4000 | 0.2000 | 0.6670 | 0.4000 | 0.5000 | Backdoor |
| 0.8000 | 0.6000 | 0.5710 | 0.8000 | 0.6670 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.9000 | 0.5000 | 0.6430 | 0.9000 | 0.7500 | Backdoor |
| 0.5000 | 0.1000 | 0.8330 | 0.5000 | 0.6250 | Nonmal |

Table B.2: Backdoor-Nonmal: Results by Class

The results by class shown in Table B.2 are interesting, because it appears from the Precision values that J4.8 is better at classifying class Backdoor, whereas Naive Bayes (which is a better classifier overall) is better at classifying Nonmal, though the F-measure would tend to suggest that both classifiers are in fact better at classifying Nonmal.

## B.1.2 Trojan-Nonmal

Table B.3 gives the results from running the J4.8 and Naive Bayes classifiers on Trojan-Nonmal.

| **Trojan-Nonmal** | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 15 (75.0000%) | 11 (55.0000%) |
| Incorrectly Classified Instances | 5 (25.0000%) | 9 (45.0000%) |
| Kappa statistic | 0.5000 | 0.1000 |
| Mean absolute error | 0.2375 | 0.4500 |
| Root mean squared error | 0.4538 | 0.6708 |
| Relative absolute error | 47.5000% | 90.0000% |
| Root relative squared error | 90.7530% | 134.1641% |
| Total number of instances | 20 | 20 |

Table B.3: Trojan-Nonmal

Here it is seen that, in contrast to Backdoor-Nonmal, J4.8 is achieving the highest proportion of correct classifications. The percentage difference between the classifiers is about the same, though (here J4.8 achieves 20% more correct classifications than Naive Bayes whereas on Backdoor-Nonmal J4.8 achieved 20% *fewer* correct classifications). Agreement as measured by the Kappa statistic is higher for J4.8 (0.5) than it was for Naive Bayes in Backdoor-Nonmal, whereas the Kappa for Naive Bayes here is lower than that for J4.8 on Backdoor-Nonmal (0.3).

| **J4.8** | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.8000 | 0.3000 | 0.7270 | 0.8000 | 0.7620 | Trojan |
| 0.7000 | 0.2000 | 0.7780 | 0.7000 | 0.7370 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.8000 | 0.7000 | 0.5330 | 0.8000 | 0.6400 | Trojan |
| 0.3000 | 0.2000 | 0.6000 | 0.3000 | 0.4000 | Nonmal |

Table B.4: Trojan-Nonmal: Results by Class

From Table B.4 it can be seen that the Precision and F-Measure for both classes in J4.8 is very close, whereas Naive Bayes seems to show a decided bias towards the Trojan class.

### B.1.3 Worm-Nonmal

Table B.5 gives the results from running the J4.8 and Naive Bayes classifiers on Worm-Nonmal.

| Worm-Nonmal | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 14 (70.0000%) | 13 (65.0000%) |
| Incorrectly Classified Instances | 6 (30.0000%) | 7 (35.0000%) |
| Kappa statistic | 0.4000 | 0.3000 |
| Mean absolute error | 0.3300 | 0.3500 |
| Root mean squared error | 0.5395 | 0.5916 |
| Relative absolute error | 66.0000 | 70.0028%% |
| Root relative squared error | 107.9094 | 118.3216%% |
| Total number of instances | 20 | 20 |

Table B.5: Worm-Nonmal

In contrast with Trojan-Nonmal, here the difference in percentage of correctly classified instances is much closer (5% as opposed to 20%). J4.8 still outperforms Naive Bayes, though less so than in Trojan-Nonmal, since J4.8 is performing about 5% worse in this comparison and Naive Bayes is performing about 10% better.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.8000 | 0.4000 | 0.6670 | 0.8000 | 0.7270 | Worm |
| 0.6000 | 0.2000 | 0.7500 | 0.6000 | 0.6670 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.4000 | 0.1000 | 0.8000 | 0.4000 | 0.5330 | Worm |
| 0.9000 | 0.6000 | 0.6000 | 0.9000 | 0.7200 | Nonmal |

Table B.6: Worm-Nonmal: Results by Class

Looking at Table B.6, it is interesting to note that J4.8 seems to favour class Worm, though not as much (0.72 to 0.66) as Naive Bayes favours Nonmal (0.72 to 0.53).

# B.2 Experiment Two

## B.2.1 Backdoor-Worm

Table B.7 gives the results from running the J4.8 and Naive Bayes classifiers on Backdoor-Worm.

| Backdoor-Worm | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 10 (50.0000%) | 12 (60.0000%) |
| Incorrectly Classified Instances | 10 (50.0000%) | 8 (40.0000%) |
| Kappa statistic | 0.0000 | 0.2000 |
| Mean absolute error | 0.4972 | 0.4000 |
| Root mean squared error | 0.6661 | 0.6325 |
| Relative absolute error | 99.4444% | 80.0000% |
| Root relative squared error | 133.2175% | 126.4911% |
| Total number of instances | 20 | 20 |

Table B.7: Backdoor-Worm

It is immediately apparent that J4.8 has completely failed here, achieving results identical to chance (which is why the Kappa statistic is 0). Naive Bayes performs slightly better, but still worse than on any of the previously-seen datasets.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.2000 | 0.2000 | 0.5000 | 0.2000 | 0.2860 | Backdoor |
| 0.8000 | 0.8000 | 0.5000 | 0.8000 | 0.6150 | Worm |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.9000 | 0.7000 | 0.5630 | 0.9000 | 0.6920 | Backdoor |
| 0.3000 | 0.1000 | 0.7500 | 0.3000 | 0.4290 | Worm |

Table B.8: Backdoor-Worm: Results by Class

It almost seems superfluous to comment on the J4.8 results in Table B.8, but there seems to be a bias in favour of Worm (Recall 0.8 to 0.2), whereas Naive Bayes prefers Backdoor (Recall 0.9 to 0.3).

## B.2.2 Trojan-Backdoor

Table B.9 gives the results from running the J4.8 and Naive Bayes classifiers on Trojan-Backdoor.

| **Trojan-Backdoor** | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 13 (65.0000%) | 15 (75.0000%) |
| Incorrectly Classified Instances | 7 (35.0000%) | 5 (25.0000%) |
| Kappa statistic | 0.3000 | 0.5000 |
| Mean absolute error | 0.3940 | 0.2500 |
| Root mean squared error | 0.5793 | 0.5000 |
| Relative absolute error | 78.7929% | 50.0000% |
| Root relative squared error | 115.8597% | 100.0000% |
| Total number of instances | 20 | 20 |

Table B.9: Trojan-Backdoor

Here is another case of Naive Bayes outperforming J4.8, though here the difference is only 10%. J4.8 still does adequately, though the Kappa statistic is down at 0.3 compared to Naive Bayes' 0.5.

| **J4.8** | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.6000 | 0.3000 | 0.6670 | 0.6000 | 0.6320 | Backdoor |
| 0.7000 | 0.4000 | 0.6360 | 0.7000 | 0.6670 | Trojan |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.6000 | 0.1000 | 0.8570 | 0.6000 | 0.7060 | Backdoor |
| 0.9000 | 0.4000 | 0.6920 | 0.9000 | 0.7830 | Trojan |

Table B.10: Trojan-Backdoor: Results by Class

If the F-measure is any indication, both classes seem to be relatively similar (in both J4.8 and Naive Bayes the difference is in the second decimal place, though the F-measures for Naive Bayes are higher by at least 0.1).

### B.2.3 Trojan-Worm

Table B.11 gives the results from running the J4.8 and Naive Bayes classifiers on Trojan-Worm.

| Trojan-Worm | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 13 (65.0000%) | 11 (55.0000%) |
| Incorrectly Classified Instances | 7 (35.0000%) | 9 (45.0000%) |
| Kappa statistic | 0.3000 | 0.1000 |
| Mean absolute error | 0.3500 | 0.4500 |
| Root mean squared error | 0.5373 | 0.6708 |
| Relative absolute error | 70.0000% | 90.0000% |
| Root relative squared error | 107.4609% | 134.1641% |
| Total number of instances | 20 | 20 |

Table B.11: Trojan-Worm

Here, J4.8 performs identically to Trojan-Backdoor (65% in both cases and with identical Kappa statistic, though the error values are slightly lower). Strangely, Naive Bayes performs 20% worse than it did on Trojan-Backdoor.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.8000 | 0.5000 | 0.6150 | 0.8000 | 0.6960 | Trojan |
| 0.5000 | 0.2000 | 0.7140 | 0.5000 | 0.5880 | Worm |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.9000 | 0.8000 | 0.5290 | 0.9000 | 0.6670 | Trojan |
| 0.2000 | 0.1000 | 0.6670 | 0.2000 | 0.3080 | Worm |

Table B.12: Trojan-Worm: Results by Class

Table B.12 indicates that for J4.8, although the Worm class has greater Precision, the F-measure is greater for the Trojan class. This is also true of Naive Bayes, though the F-measure for Worm is much less here than for J4.8, that for Trojan being equal within 1 decimal place.

# B.3 Experiment Three

## B.3.1 Backdoor-Trojan-Worm

Table B.13 gives the results from running the J4.8 and Naive Bayes classifiers on Backdoor-Trojan-Worm.

| **Backdoor-Trojan-Worm** | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 10 (33.3333%) | 15 (50.0000%) |
| Incorrectly Classified Instances | 20 (66.6667%) | 15 (50.0000%) |
| Kappa statistic | 0.0000 | 0.2500 |
| Mean absolute error | 0.4607 | 0.3333 |
| Root mean squared error | 0.6447 | 0.5774 |
| Relative absolute error | 103.6667% | 75.0000% |
| Root relative squared error | 136.7599% | 122.4745% |
| Total number of instances | 30 | 30 |

Table B.13: Backdoor-Trojan-Worm

In the worst results yet, J4.8 manages to classify only 33% of cases correctly, whereas Naive Bayes - outperforming it - performs no better than chance (though the Kappa statistic for Naive Bayes may indicate that the accuracy of the model is in fact slightly greater than these results suggest).

| **J4.8** | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5000 | 0.3000 | 0.4550 | 0.5000 | 0.4760 | Backdoor |
| 0.3000 | 0.3500 | 0.3000 | 0.3000 | 0.3000 | Trojan |
| 0.2000 | 0.3500 | 0.2220 | 0.2000 | 0.2110 | Worm |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5000 | 0.2500 | 0.5000 | 0.5000 | 0.5000 | Backdoor |
| 0.8000 | 0.4000 | 0.5000 | 0.8000 | 0.6150 | Trojan |
| 0.2000 | 0.1000 | 0.5000 | 0.2000 | 0.2860 | Worm |

Table B.14: Backdoor-Trojan-Worm: Results by Class

Table B.14 would seem to show that for J4.8 the Precision and Recall values follow each other, being highest for Backdoor and lowest for Worm. For Naive Bayes the Precision values are of course the same for each class, but the Recall is higher for Trojan, which influences the F-measure.

### B.3.2 Backdoor-Worm-Nonmal

Table B.15 gives the results from running the J4.8 and Naive Bayes classifiers on Backdoor-Worm-Nonmal.

| **Backdoor-Worm-Nonmal** | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 10 (33.3333%) | 15 (50.0000%) |
| Incorrectly Classified Instances | 20 (66.6667%) | 15 (50.0000%) |
| Kappa statistic | 0.0000 | 0.2500 |
| Mean absolute error | 0.4281 | 0.3333 |
| Root mean squared error | 0.6131 | 0.5774 |
| Relative absolute error | 96.3254% | 75.0000% |
| Root relative squared error | 130.0683% | 122.4745% |
| Total number of instances | 30 | 30 |

Table B.15: Backdoor-Worm-Nonmal

These results are virtually identical to those for Backdoor-Trojan-Worm (in the case of Naive Bayes they are exactly identical, whereas some of the error values differ in J4.8). Neither classifier can achieve better than chance, and J4.8 achieves significantly less well.

| **J4.8** | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.3000 | 0.4000 | 0.2730 | 0.3000 | 0.2860 | Backdoor |
| 0.5000 | 0.2500 | 0.5000 | 0.5000 | 0.5000 | Worm |
| 0.2000 | 0.3500 | 0.2220 | 0.2000 | 0.2110 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.9000 | 0.5500 | 0.4500 | 0.9000 | 0.6000 | Backdoor |
| 0.2000 | 0.0500 | 0.6670 | 0.2000 | 0.3080 | Worm |
| 0.4000 | 0.1500 | 0.5710 | 0.4000 | 0.4710 | Nonmal |

Table B.16: Backdoor-Worm-Nonmal: Results by Class

Unlike Table B.15, the results in Table B.16 are not identical to those from the previous page (Backdoor-Trojan-Worm). In J4.8 the Precision, Recall, and F-Measure for class Worm are all equal, and for the other two classes these three measurements are very close. Naive Bayes has highest Precision for class Worm, but highest F-Measure for class Backdoor.

### B.3.3 Trojan-Backdoor-Nonmal

Table B.17 gives the results from running the J4.8 and Naive Bayes classifiers on Trojan-Backdoor-Nonmal.

| Trojan-Backdoor-Nonmal | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 18 (60.0000%) | 16 (53.3333%) |
| Incorrectly Classified Instances | 12 (40.0000%) | 14 (46.6667%) |
| Kappa statistic | 0.4000 | 0.3000 |
| Mean absolute error | 0.2986 | 0.3111 |
| Root mean squared error | 0.4889 | 0.5578 |
| Relative absolute error | 67.1944% | 70.0000% |
| Root relative squared error | 103.7145% | 118.3216% |
| Total number of instances | 30 | 30 |

Table B.17: Trojan-Backdoor-Nonmal

Unlike the previous two comparisons in Experiment Three, both classifiers are now performing better than chance, and J4.8 once again beating Naive Bayes. However, the Kappa values do not indicate particularly good agreement.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.4000 | 0.0000 | 1.0000 | 0.4000 | 0.5710 | Backdoor |
| 0.7000 | 0.3000 | 0.5380 | 0.7000 | 0.6090 | Trojan |
| 0.7000 | 0.3000 | 0.5380 | 0.7000 | 0.6090 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5000 | 0.1500 | 0.6250 | 0.5000 | 0.5560 | Backdoor |
| 0.7000 | 0.4000 | 0.4670 | 0.7000 | 0.5600 | Trojan |
| 0.4000 | 0.1500 | 0.5710 | 0.4000 | 0.4710 | Nonmal |

Table B.18: Trojan-Backdoor-Nonmal: Results by Class

The J4.8 section of Table B.18 indicates that the precision and recall for Trojan and Nonmal is identical. Furthermore, class Backdoor had no false positives, hence the Precision was 1, though since the Recall or True Positive rate was quite low, the F-measure for Backdoor was lower than that for the other classes. Naive Bayes showed similar F-Measures for Backdoor and Trojan, and a lower one for Nonmal.

### B.3.4 Trojan-Worm-Nonmal

Table B.19 gives the results from running the J4.8 and Naive Bayes classifiers on Trojan-Worm-Nonmal.

| Trojan-Worm-Nonmal | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 15 (50.0000%) | 13 (43.3333%) |
| Incorrectly Classified Instances | 15 (50.0000%) | 17 (56.6667%) |
| Kappa statistic | 0.2500 | 0.1500 |
| Mean absolute error | 0.3330 | 0.3778 |
| Root mean squared error | 0.5531 | 0.6146 |
| Relative absolute error | 74.9167% | 85.0000% |
| Root relative squared error | 117.3350% | 130.3840% |
| Total number of instances | 30 | 30 |

Table B.19: Trojan-Worm-Nonmal

It is interesting that out of the four comparisons in Experiment Three, only in Backdoor-Trojan-Nonmal were either of the classifiers able to achieve better results than chance. Unfortunately in this case the losing trend has resumed, with J4.8 performing at chance and Naive Bayes performing just less. Kappa statistics of 0.25 and 0.15 might indicate that there is slightly more agreement than these disappointing results would suggest, but even this is not much.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5000 | 0.3000 | 0.4550 | 0.5000 | 0.4760 | Trojan |
| 0.4000 | 0.3000 | 0.4000 | 0.4000 | 0.4000 | Worm |
| 0.6000 | 0.1500 | 0.6670 | 0.6000 | 0.6320 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.8000 | 0.6500 | 0.3810 | 0.8000 | 0.5160 | Trojan |
| 0.2000 | 0.0500 | 0.6670 | 0.2000 | 0.3080 | Worm |
| 0.3000 | 0.1500 | 0.5000 | 0.3000 | 0.3750 | Nonmal |

Table B.20: Trojan-Worm-Nonmal: Results by Class

Table B.20 shows F-Measure values for Trojan and Worm as fairly close, with Nonmal significantly higher. Naive Bayes has close F-Measures for Worm and Nonmal (around 0.3-0.4) with Trojan higher (0.5160).

# B.4 Experiment Four

## B.4.1 Part One: Backdoor-Trojan-Worm-Nonmal

Table B.21 gives the results from running the J4.8 and Naive Bayes classifiers on all four classes at once.

| Backdoor-Trojan-Worm-Nonmal | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 19 (47.5000%) | 17 (42.5000%) |
| Incorrectly Classified Instances | 21 (52.5000%) | 23 (57.5000%) |
| Kappa statistic | 0.3000 | 0.2333 |
| Mean absolute error | 0.2828 | 0.2875 |
| Root mean squared error | 0.4978 | 0.5358 |
| Relative absolute error | 75.4167% | 76.6667% |
| Root relative squared error | 114.9733% | 123.7288% |
| Total number of instances | 40 | 40 |

Table B.21: Backdoor-Trojan-Worm-Nonmal

Given the trends seen in Experiment Three, and the general effect of the "Curse of Dimensionality" on the results, it is perhaps not surprising that these results are so bad here. neither classifier can even achieve chance accuracy. Even so, J4.8 outperforms Naive Bayes, though only by 5%.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5000 | 0.2330 | 0.4170 | 0.5000 | 0.4550 | Backdoor |
| 0.6000 | 0.1330 | 0.6000 | 0.6000 | 0.6000 | Trojan |
| 0.4000 | 0.2330 | 0.3640 | 0.4000 | 0.3810 | Worm |
| 0.4000 | 0.1000 | 0.5710 | 0.4000 | 0.4710 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5000 | 0.2000 | 0.4550 | 0.5000 | 0.4760 | Backdoor |
| 0.7000 | 0.4330 | 0.3500 | 0.7000 | 0.4670 | Trojan |
| 0.2000 | 0.0330 | 0.6670 | 0.2000 | 0.3080 | Worm |
| 0.3000 | 0.1000 | 0.5000 | 0.3000 | 0.3750 | Nonmal |

Table B.22: Backdoor-Trojan-Worm-Nonmal: Results by Class

Table B.22 shows that for J4.8 the classes in order of decreasing F-Measure are Trojan, Nonmal, Backdoor, Worm, whereas for Naive Bayes they are Worm, Nonmal, Trojan,Backdoor.

### B.4.2 Part Two: Mal-Nonmal

| Mal-Nonmal | | |
|---|---|---|
| **Item** | **J4.8** | **Naive Bayes** |
| Correctly Classified Instances | 29 (72.5000%) | 24 (60.0000%) |
| Incorrectly Classified Instances | 11 (27.5000%) | 16 (40.0000%) |
| Kappa statistic | 0.2903 | 0.2381 |
| Mean absolute error | 0.2725 | 0.4000 |
| Root mean squared error | 0.5022 | 0.6325 |
| Relative absolute error | 71.4138% | 104.8274% |
| Root relative squared error | 115.9301% | 145.9917% |
| Total number of instances | 40 | 40 |

Table B.23: Mal-Nonmal

On dispensing with the distinctions between different types of malware, and thus lifting the Curse of Dimensionality, accuracy shoots back up again. However, the Kappa statistics still do not look too healthy, even J4.8, as it achieves 72.5% correct classification, can only manage 0.2903, little better than Naive Bayes' 0.2381.

| J4.8 | | | | | |
|---|---|---|---|---|---|
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.8000 | 0.5000 | 0.8280 | 0.8000 | 0.8140 | Mal |
| 0.5000 | 0.2000 | 0.4550 | 0.5000 | 0.4760 | Nonmal |
| **Naive Bayes** | | | | | |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **Class** |
| 0.5330 | 0.2000 | 0.8890 | 0.5330 | 0.6670 | Mal |
| 0.8000 | 0.4670 | 0.3640 | 0.8000 | 0.5000 | Nonmal |

Table B.24: Mal-Nonmal: Results by Class

From Table B.24 it is seen that both J4.8 and Naive Bayes tend to favour Mal. This is not a surprise, as the result of amalgamating all the Malicious categories means that there are three times as many Mal cases as Nonmal ones.

## B.5 General Trends

On a class basis J4.8 tends to favour opposite or different classes from Naive Bayes. J4.8 outperforms Naive Bayes on all but a few occasions. In Experiment Three only one comparison achieved results better than chance.

Appendix C

# The Data Gathering Program

## C.1 Introduction

This appendix gives a brief description of the program that was developed to collect data on structural and behavioural attributes of programs and processes running under Windows.

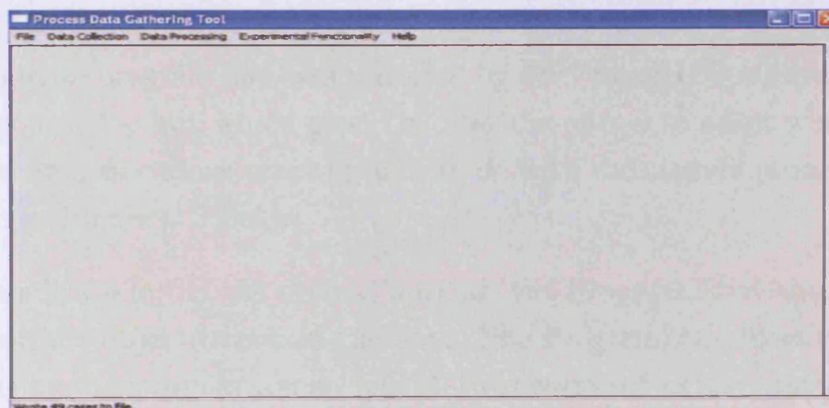



Figure C.1: The initial VMON program window

## C.2 General Design and User Interface

The program, called VMON, was written in C++ and was designed using full object orientation. Each individual element of functionality was implemented as a class. Figure C.2 shows the essential relationship between the most important classes.

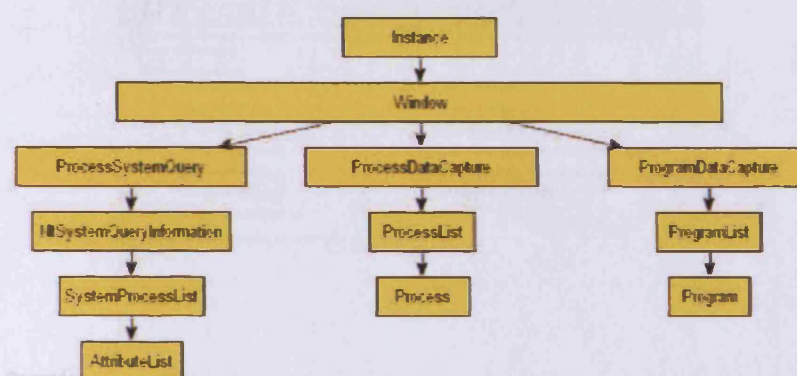



Figure C.2: Diagram of Key Classes in the VMON Program

The Window class manages the main program window, which is laid out following standard practice. The data collection and processing tasks are handled by individual classes, and there are also ancillary classes not shown in the diagram that act as wrappers for files, common open and save dialog boxes, status bars, etc.

## C.3 Collecting Structural Data

Structural data on program files was collected by the ProgramDataCapture class. This class displays a dialog box which gives the user the option to select which data items to capture, as well as various other options to do with the capture process. The dialog box is shown in Figure C.3 below.

Once the user has selected the desired options, the ProgramDataCapture object will use a ProgramList object to collect the data. The ProgramList object can recursively search through a directory structure, find all the program files it contains, and capture data on each one using a storage class called Program. Data can be output to a file or to a status window in the main program, and attribute data can be printed as well, in C4.5 or ARFF (WEKA) format.

The Program class works by mapping a program file into memory, then reading the headers and other file structures and outputting the relevant data. It is also capable of walking the import table to calculate how many functions a program imports.

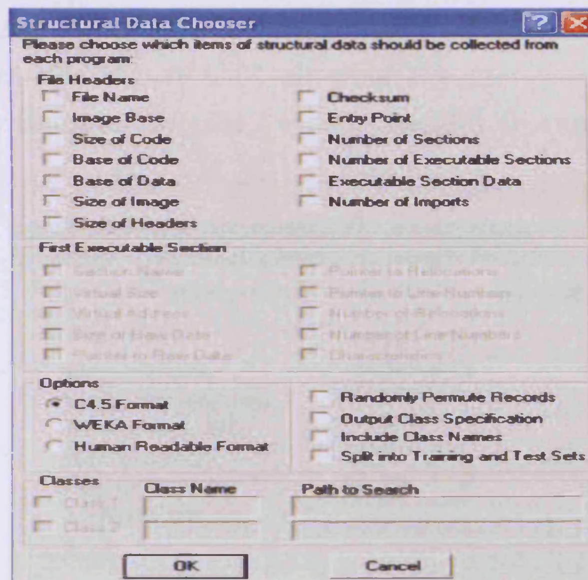



Figure C.3: The ProgramDataCapture dialog box

## C.4 Collecting Behavioural Data

Behavioural data can be collected in two ways. The ProcessDataCapture class provides a method for collecting data on individual processes in a similar manner to ProgramDataCapture. The difference is that an individual program needs to be running, and the data must be captured using the PSAPI functions. Figure C.4 illustrates the ProgramDataCapture dialog box.

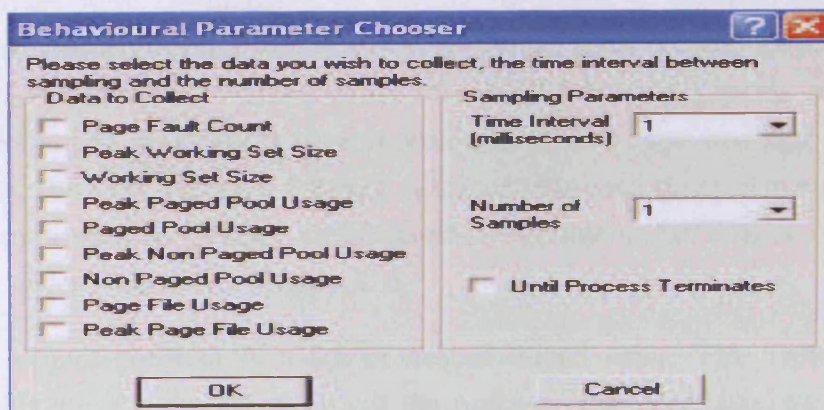



Figure C.4: The ProcessDataCapture Dialog Box

The data for the behavioural experiments was not collected in the above manner, as the amount of data provided by the PSAPI functions is quite limited. Instead, the undocumented NtSystemQueryInformation() API was used (for details, see Chapter 5). Wrapper classes called NtSystemQueryInformation and ProcessSystemQuery were

created. The outermost wrapper class is similar to ProcessDataCapture in that it displays a dialog box (see Figure C.5) allowing the user to choose which data are captured, how many samples to take, which program to run, and where the data should end up.

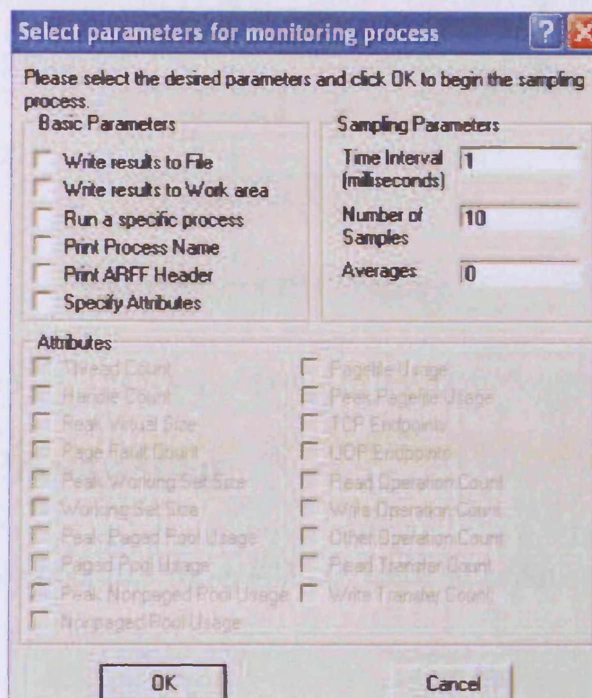



Figure C.5: The ProcessSystemQuery Dialog Box

Because samples were being taken at regular intervals, it was necessary to call the NtSystemQueryInformation() function several times. This created an interesting challenge. The API function returned a set of results for every process running on the system in sequential order each time it was called, so it was necessary to provide a way of storing the data samples for each attribute for each process in such a way that storage was organised by process rather than by sample stage. This was done using a data storage object, shown in Figure C.6.

The storage object consists of a set of nested linked lists. The SystemProcessList object contains one AttributeList object per process. The AttributeList object in turn contains a set of lists corresponding to each attribute. These internal lists hold the actual data.

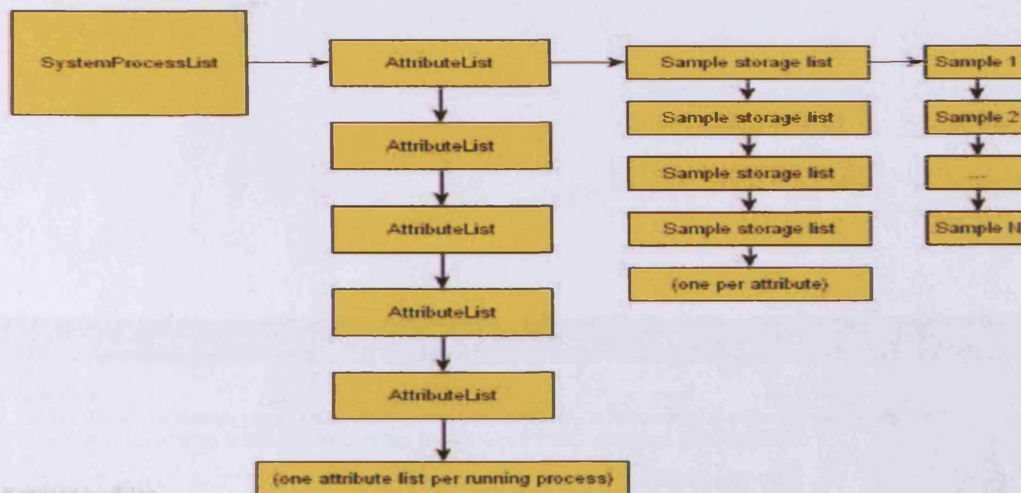


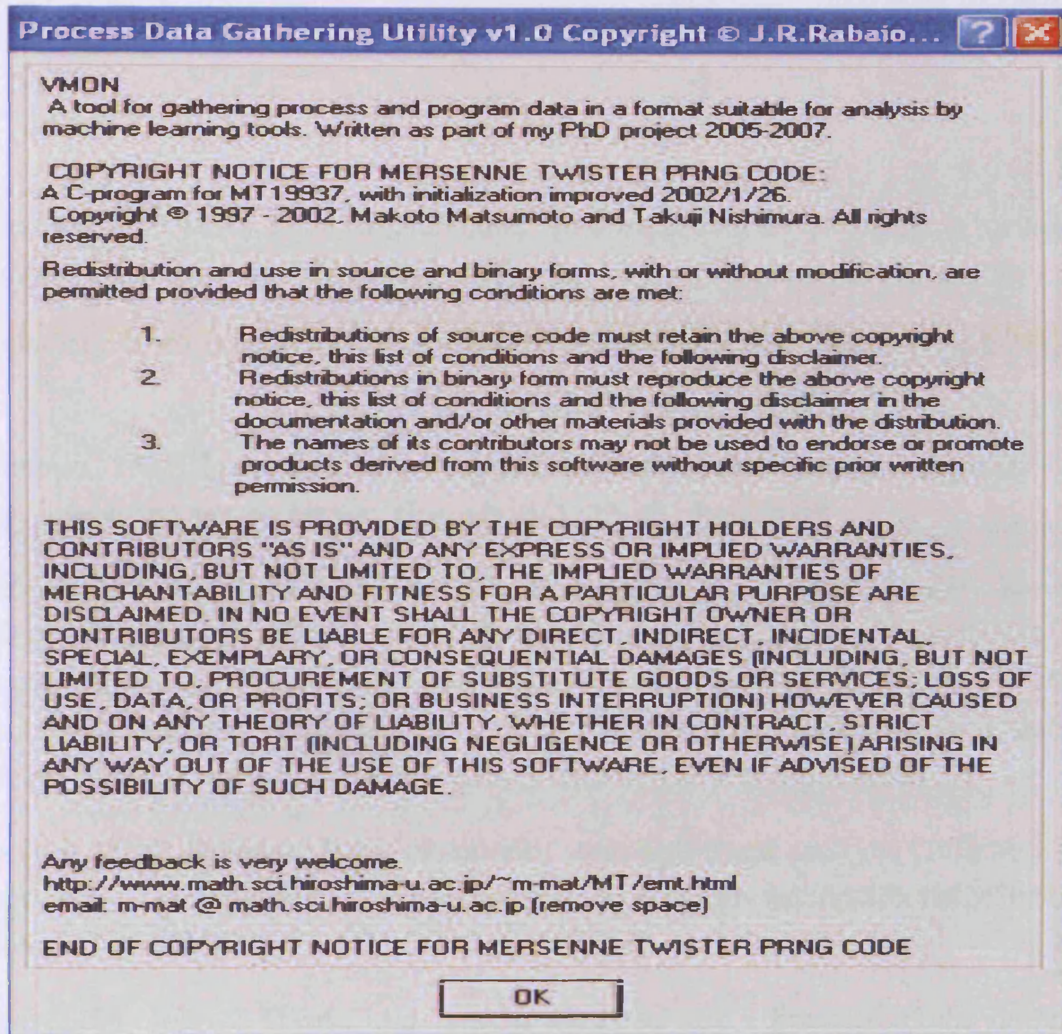

Figure C.6: Diagram of Behavioural Data Storage Object

## C.5 Data Processing

VMON also has the capability to shuffle the lines in a data file into pseudorandom order.

## C.6 Acknowledgements

Pseudorandom number generation was accomplished using an implementation of the Mersenne Twister algorithm by Makoto Matsumoto and Takuji Nishimura of Hiroshima University, who have made their source code available for use [Matsumoto and Nishimura, 1998]. Their copyright notice is reproduced in the about box figure shown below.



Process Data Gathering Utility v1.0 Copyright © J.R.Rabaio...

VMON
A tool for gathering process and program data in a format suitable for analysis by machine learning tools. Written as part of my PhD project 2005-2007.

COPYRIGHT NOTICE FOR MERSENNE TWISTER PRNG CODE:
A C-program for MT19937, with initialization improved 2002/1/26.
Copyright © 1997 - 2002, Makoto Matsumoto and Takuji Nishimura. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

END OF COPYRIGHT NOTICE FOR MERSENNE TWISTER PRNG CODE

OK

Figure C.7: The About Box, containing the Mersenne Twister copyright notice

# Bibliography

[Adleman, 1988 1990] Leonard Adleman. An abstract theory of computer viruses. *Advances in Cryptology (CRYPTO 88) - also LNCS 403*, pages 354–374, 1988 (1990).

[AlephOne, 1996] AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, 7, 1996.

[Anderson, 2003] Ross Anderson. 'trusted computing' and competition policy - issues for computing professionals. *Upgrade*, 4(3):35–41, June 2003.

[Barham *et al.*, 2003] Paul R. Barham, Boris Dragovic, Keir A. Fraser, Steven M. Hand, Timothy L. Harris, Alex C. Ho, Evangelos Kotsovinos, Anil V.S. Madhavapeddy, Rolf Neugebauer, Ian A. Pratt, and Andrew K. Warfield. Technical report number 553: Xen 2002. Technical report, University of Cambridge, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-553.pdf, 2003.

[Baseline, 2006] Baseline. Bank of America seeks anti-fraud anodyne (15th may 2006). web site, May 2006. http://www.baselinemag.com/article2/0,1540,1962470,00.asp - accessed 10/02/2007.

[Bayes, 1763] Rev. T Bayes. An essay towards solving a problem in the doctrine of chance. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763.

[Biermann and Feldman, 1972] A. Biermann and G. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Trans. Comput.*, 21:592–597, 1972.

[Bishop, 1992] Matt Bishop. An overview of computer viruses in a research environment. Technical report, Department of Mathematics and Computer Science, Dartmouth College, 1992.

[Blyth and Kovacich, 2001] Andrew Blyth and Gerald L. Kovacich. *Information Assurance: Surviving in the Information Environment.* Springer, 2001.

[Boudon, 1977] R. Boudon. *The unintended consequences of social action.* St. Martin's Press, 1977.

[Bradbury, 2006] Danny Bradbury. The metamorphosis of malware writers. *Computers & Security*, 25:89–90, 2006.

[Cam-Winget *et al.*, 2003] Nancy Cam-Winget, Russ Housley, David Wagner, and Jesse Walker. Security flaws in 802.11 data link protocols. *Communications of the ACM*, 46(5):35–39, May 2003.

[Chang and Young, 2005] David B. Chang and Carl S. Young. Infection dynamics on the Internet. *Computers & Security*, 24:280–286, 2005.

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

[Cohen, 1960] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46, 1960.

[Cohen, 1987] Dr Frederick B. Cohen. Computer viruses: Theory and experiments. *Computers & Security*, 6:22–35, 1987.

[Cohen, 1989] Dr Frederick B. Cohen. Computational aspects of computer viruses. *Computer Security*, 8(4):325–344, 1989.

[Conover, 1999] Matt Conover. w00w00 on heap overflows. Technical report, w00w00, http://www.w00w00.org/files/articles/heaptut.txt, 1999. Accessed 07/03/2007.

[Counterpane and Messagelabs, 2005] Counterpane and Messagelabs. 2005 attack trends & analysis. Technical report, Counterpane and Messagelabs, 2005.

[Crosbie and Spafford, 1995a] Mark Crosbie and Gene Spafford. Active defense of a computer system using autonomous agents. Technical report, COAST Group Dept. of Computer Sciences Purdue University, 1995.

[Crosbie and Spafford, 1995b] Mark Crosbie and Gene Spafford. Defending a computer system using autonomous agents. In *Proceedings of 8th National Information Systems Security Conference (1995)*, 1995.

[Cybenko, 1989] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signal and Systems*, 2:303–314, 1989.

[Day, 2006] Oliver Day. Analysis of Microsoft's suicide note part 1. Free Software Foundation BadVista Blog, January 2006. http://badvista.fsf.org/blog/analysis-of-microsofts-suicide-note-part-1 - accessed 06/03/2007.

[Deering and Hinden, 1995] S. Deering and R. Hinden. RFC 1883 - internet protocol, version 6 (IPv6) specification. 1995.

[Eck, 1985] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4:269–286, 1985.

[Edelman, 2002] Benjamin G. Edelman. Declaration of Benjamin G. Edelman, WASHINGTONPOST.NEWSWEEK, LLC. ET AL v THE GATOR CORPORATION, June 2002. http://cyber.law.harvard.edu/people/edelman/pubs/gator-062502.pdf - accessed 07/03/2007.

[Evers, 2006] Joris Evers. Windows defense handcuffs good guys. CNet web site, August 2006. http://news.com.com/Windows+defense+handcuffs+good+guys/2100-7355_3-6104379.html, accessed 06/03/2007.

[Finextra, 2006] Finextra. UK phishing fraud losses double (7th march 2006). Finextra (web site), March 2006. http://www.finextra.com/fullstory.asp?id=15013 - accessed 08/02/2007.

[Flake, 2006] Halvar Flake. More on automated malware classification and naming. April 2006. http://addxorrol.blogspot.com/2006/04/more-on-automated-malware.html - accessed 07/06/2006.

[Fleiss, 1981] Joseph L. Fleiss. *Statistical methods for rates and proportions.* 2ed. John Wiley & Sons Inc., New York, 1981.

[Forrest *et al.*, 1994] Stephanie Forrest, Lawrence Allen, Alan S Perelson, and Rajesh Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of 1994 IEEE Symposium on Research in Security and Privacy*, 1994.

[Forrest *et al.*, 1996] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for UNIX processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[Gator, 2005] Gator. Gator ewallet - the smart online companion, March 2005. http://www.gator.com/home2.html - accessed 23/03/2005.

[Gollmann, 1999] Dieter Gollmann. *Computer Security*. Wiley, 1999.

[Goring *et al.*, 2007] S. P. Goring, J. R. Rabaiotti, and Antonia J. Jones. Anti-keylogging measures for secure internet login: an example of the law of unintended consequences. *Computers & Security*, 2007. (In Press).

[Guardian, 2006] The Guardian. Corrections and clarifications (21st september 2006), September 2006. http://www.guardian.co.uk/corrections/story/0,,1877068,00.html - accessed 14/10/2006.

[Gunn, 1974] J.B. Gunn. Use of virus functions to provide a virtual APL interpreter under user control. *ACM*, pages 163–168, 1974.

[Gutmann, 2007] Peter Gutmann. A cost analysis of windows vista content protection. Technical report, University of Auckland, 2007.

[Hart, 2005] Johnson M. Hart. *Windows System Programming*. Addison Wesley, 2005.

[Hornik *et al.*, 1989] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:356–366, 1989.

[Hückmann, 2006] Daniel Hückmann. Running a desktop with full system privileges. Technical report, Pandora Security, 2006.

[Hunt and Brubacher, 1999] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[Hyafil and Rivest, 1976] L. Hyafil and R. Rivest. Constructing optimal binary decision trees is NP-complete. *Inform. Process. Lett.*, 5:15–17, 1976.

[Johnson and Cobain, 2006] Bobbie Johnson and Ian Cobain. Security flaw leaves 3m HSBC online accounts open to fraud (10th august 2006). Newspaper Article, August 2006. http://business.guardian.co.uk/story/0,,1841853,00.html, accessed 16/02/2007.

[Kdm, 2005] Kdm. Analysis of a win32 userland rootkit. Technical report, http://www.securiteam.com/securityreviews/5FP0E0AGAC.html, July 2005. Accessed 28/03/2007.

[Kephart *et al.*, 1993] Jeffrey O. Kephart, David M. Chess, and Steve R. White. Computers and epidemiology. *IEEE SPECTRUM*, 1993.

[Kernighan and Ritchie, 1988] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (second edition)*. Prentice Hall, 1988.

[Kim *et al.*, 2004] C. Kim, W. Kim, and M. Hong. Effective detector set generation and evolution for artificial immune system. In *International Conference on Computational Science*, volume 3037, pages 491–498. Springer-Verlag, 2004.

[Knuth, 1997] Donald Knuth. *The Art of Computer Programming (third edition)*, volume 2. Addison Wesley, third edition, 1997.

[Kuster, 2003] Robert Kuster. Three ways to inject your code into another process. Technical report, CodeGuru, http://www.codeguru.com/Cpp/W-P/system/processesmodules/article.php/c5767, July 2003. Accessed 28/03/2006.

[Lee and Mody, 2006] Tony Lee and Jigar J. Mody. Behavioural classification. EICAR, May 2006.

[Lee *et al.*, 2004] Hyungjoon Lee, Wonil Kim, and Manpyo Hong. Artificial immune system against viral atttack. In *International Conference on Computational Science*, volume 3037, pages 499–506. Springer-Verlag, 2004.

[Lemos, 2006] Robert Lemos. Researchers eye machines to tackle malware, 2006. http://www.theregister.co.uk/2006/06/10/machines_analyse_malware/ - accessed 07/06/2006.

[Ley, 2005] Florida man sues bank over $90k wire fraud (8th february 2005). The Register, February 2005. http://www.theregister.co.uk/2005/02/08/e-banking/_trojan_lawsuit/ - accessed 08/02/2007.

[Leyden, 2005] John Leyden. Trojan phishing suspect hauled in (4th april 2005). The Register (web site), April 2005. http://www.theregister.co.uk/2005/04/04/estonian_trojan_suspect_cuffed/ - accessed 11/02/2007.

[Lhee and Chapin, 2003] Kyung-Suk Lhee and Steve J Chapin. Buffer overflow and format string overflow vulnerabilities. *Software - Practice and Experience*, 33:423–460, 2003.

[Linde, 1975] R. R. Linde. Operating system penetration. In *AIFIPS National Computer Conference*, 1975.

[LURHQ, 2004] LURHQ. Win32.Grams E-Gold Account Siphoner Analysis (4th november 2004). web site, November 2004. http://www.lurhq.com/grams.html - accessed 10/02/2007.

[Matsumoto and Nishimura, 1998] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[Microsoft, 2005] Microsoft. Official xbox website. Web Page, 2005. http: www.xbox.com - accessed 07/03/2007.

[Microsoft, 2006] Microsoft. *Microsoft Portable Executable and Common Object File Format Specification.* Microsoft Corporation, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, 8.0 edition, 2006.

[Microsoft, 2007] Microsoft. Windows vista: Learn about the features: Safer. Web site, 2007. http://www.microsoft.com/windows/products/windowsvista/features/safer.mspx -Accessed 07/03/2007.

[Perriot, 2007] Frederic Perriot. W32.welchia.worm. Technical report, Symantec Corporation, 2007.

[Petzold, 1999] Charles Petzold. *Programming Windows (fifth edition).* Microsoft Press, fifth edition, 1999.

[Pietrek, 1994 updated 2002] Matt Pietrek. Peering inside the pe: A tour of the win32 portable executable file format. *Microsoft Systems Journal (MSDN Magazine)*, 1994 (updated 2002.

[Pietrek, 2002] Matt Pietrek. Inside windows: An in-depth look into the win32 portable executable file format. *MSDN Magazine*, 17(3), February and March 2002.

[Probert, 1994] Matthew Probert. The virus researcher handbook. Technical report, Servile Software, 1994.

[Quinlan, 1989] J. R. Quinlan. Unknown attribute values in induction. In *Proceedings of Sixth International Machine Learning Workshop*, 1989.

[Quinlan, 1993a] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[Quinlan, 1993b] J.R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of Tenth International Conference on Machine Learning*, pages 234–243. Morgan Kaufman, 1993.

[Rector and Newcomer, 1997] Brent E. Rector and Joseph M. Newcomer. *Win32 Programming*. Addison Wesley, 1997.

[Richardson, 2005] Tim Richardson. Brits fall prey to phishing (3rd may 2005). The Register (web site), May 2005. http://www.theregister.co.uk/2005/05/03/aol_phishing/ - accessed 08/02/2007.

[Richter, 1995] Jeffrey Richter. *Advanced Windows*. Microsoft Press, 1995.

[Russinovich and Solomon, 2005] Mark Russinovich and David Solomon. *Microsoft Windows Internals (fourth edition)*. Microsoft Press, 4th edition, 2005.

[Russinovich, 2005] Mark Russinovich. Sony, rootkits and digital rights management gone too far. Technical report, Sysinternals, 2005.

[Rutkowska, 2006] Joanna Rutkowska. Introducing blue pill (22 june 2006). The official blog of the invisiblethings.org, June 2006. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html - accessed 11/02/2007.

[Sabin, 2004] Todd Sabin. Comparing binaries with graph isomorphisms. Technical report, BindView RAZOR Team, http://www.bindview.com/Services/Razor/Papers/2004/comparing_binaries.cfm, 2004. Accessed 07/07/2006.

[Safford, 2002] David Safford. Clarifying misinformation on tcpa. Technical report, IBM, http://www.research.ibm.com/gsal/tcpa/tcpa_rebuttal.pdf, October 2002. Accessed 08/03/2007.

[Salomon, 2006] David Salomon. *Foundations of Computer Security*. Springer, 2006.

[Scape, 2004] Scape. Metasploit's meterpreter. Technical report, http://www.metasploit.com/projects/Framework/docs/meterpreter.pdf, December 2004. Accessed 11/02/2007.

[Schneier, 2005] Bruce Schneier. Real story of the rogue rootkit. *Wired.com*, November 2005. http://www.wired.com/politics/security/commentary/securitymatters/2005/11/69601 - accessed 29/05/2007.

[Schultz *et al.*, 2001] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[Schwartau, 1994] Winn Schwartau. *Information Warfare.* 1st edition, 1994.

[Shin and Spears, 2006] Jinwook Shin and Diana F. Spears. The basic building blocks of malware. Technical report, University of Wyoming, http://www.cs.uwyo.edu/ dspears/papers/bbb.pdf, 2006. Accessed 08/03/2007.

[Shoch and Hupp, 1982] J.F. Shoch and J.A. Hupp. The 'worm' programs - early experience with a distributed computation. In *CACM*, 1982.

[Stallman, 2002] Richard Stallman. *Free Software, Free Society*, chapter Can You Trust Your Computer? The GNU Project, 2002.

[Stasiukonis, 2006] Steve Stasiukonis. Social engineering the usb way. Technical report, Secure Network Technologies Inc, http://www.darkreading.com/document.asp?doc_id=95556&WT.svl=column1_1, 2006. Accessed 27/06/2006.

[Stoll, 1989] Clifford Stoll. *The Cuckoo's Egg.* Pan Books, 1989.

[Stroustrup, 1997] Bjarne Stroustrup. *The C++ Programming Language (third edition).* Addison Wesley, 1997.

[Szor, 2005] Peter Szor. *The Art of Computer Virus Research and Defense.* Symantec Press (Addison-Wesley), 2005.

[Tan *et al.*, 2006] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison-Wesley, 2006.

[Tesauro *et al.*, 1996] Gerald Tesauro, Jeffrey O. Kephart, and Gregory B. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5–6, 1996.

[Thurston, 2007] Richard Thurston. Tk maxx owner criticised after security breach. Yahoo! News, January 2007. http://uk.news.yahoo.com/30012007/152/tk-maxx-owner-criticised-security-breach.html - accessed 08/02/2007.

[VMWare, 2007] VMWare. Microsoft virtualization licensing and distribution terms. Technical report, VMWare Corporation, http://www.vmware.com/solutions/whitepapers/msoft_licensing_wp.html, 2007. Accessed 30/05/2007.

[VX, 2006] *VX heavens website.* http://vx.netlux.org/, 2006. Accessed 05/12/2006.

[Watkins, 2000] Andrew Watkins. An immunological approach to intrusion detection. In *Proceedings of 12th Annual Canadian Information Technology Security Symposium*, 2000.

[Webb, 2005] Bill Webb. Foistware / spyware - gator, offercompanion, trickler, GAIN. Technical report, http://www.cexx.org/gator.htm, September 2005. Accessed 07/10/2007.

[White *et al.*, 1999] Steve R White, Morton Swimmer, Edward J. Pring, William C. Arnold, David M. Chess, and John F. Morar. Anatomy of a commercial-grade immune system. In *International Virus Bulletin Conference*, 1999.

[Wikipedia, 2007] Wikipedia. Phishing. Wikipedia article, 2007. http://en.wikipedia.org/wiki/Phishing - accessed 08/02/2007.

[Witten and Frank, 2005] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufman, June 2005.

[Zuo and Zhou, 2004] Zhihong Zuo and Mingtian Zhou. Some further theoretical results about computer viruses. *The Computer Journal*, 47(6):627–633, 2004.

[Zuo *et al.*, 2006] Zhihong Zuo, Qingxin Zhu, and Mingtian Zhou. Infection, imitation and a hierarchy of computer viruses. *Computers & Security*, 25:469–473, 2006.

# Index


Activity Monitors, 41
Adware, 30, 93
  identification criteria, 31
Anomaly Detection, 66, 133
  combining with authentication, 134
API call monitoring, 103
Authentication
  combining with anomaly detection, 134

Backdoor, 29
Behavioural Classification, 42, 50, 121–128
Blaster worm, 27
Blue Pill, 64
Botnet, 32, 34
Bots, 32
Buffer overflow, 36

C4.5, 71–73
  C4.5Rules, 72
  Example Input Files, 72
Call-to-Pop Trick, 84
Classifier, 66
Code Injection, 36
Computer Misuse Act 1990, 44
Connectivity, 15
Cross-Validation, 72, 124, 159
Curse of Dimensionality, 118

Data Execution Prevention, 81
Data theft, 94
Debuggers (for Process Monitoring), 102
Decision Trees, 67–68
Device Drivers (for Process Monitoring), 103
Distributed Denial of Service Attacks, 32
DLL Injection, 103
Drive-by Download, 31

E-Gold, 63
Email, 28
Email worms, 27
Emulators, 99
Encryption, 43
Entry Point Obscuring, 80
Executable Authentication Schemes, 133

File Header(PE), 79
Format String Exploits, 37
FSM induction, 131

Gathering Behavioural Data, 98
Gator eWallet, 31
GCC, 54

Heap Overflow, 37
Heuristic Scanners, 40

IBM Immune System, 48
IFF, 132
Implicit linking, 82

Import Address Table (IAT), 83
Imported functions, 82
Impressions of the Malware Research Field, 130
Incremental Induction, 120
Information Warfare, 16–18
Internet Worm, the, 27
Intrusion Detection Systems(IDS), 18

Kappa Statistic, 111
Keystroke Loggers, 57

Law of Unintended Consequences, 56
Learning Curves (structural classification), 111
Love Bug, 28

Machine Learning, 66–75
Malware, 22
- as weapons, 17
- backdoors, 29
  - Asylum, 89
- botnets, 32, 34
- detection countermeasures, 42
  - concealment, 43
  - encryption and packing, 43
  - polymorphism and metamorphism, 43
  - retroviruses, 44
- detection methods, 38
  - activity monitors, 41
  - agent-based systems, 50
  - analysis strategies, 48
  - behaviour-based detection, 42
  - heuristic scanners, 40
  - practical systems, 48
  - signature scanners, 40
  - structure-based detection, 41
- legality of, 44
- naming conventions, 37
  - CARO, 38
- obtaining research samples, 106
- overview of combat approaches, 19
- payloads, 35
- rootkits, 35, 103
- spreading via email, 28
- spyware
  - drive-by download, 31
- spyware and adware, 30
  - criteria, 31
  - Gator, 31
- Trojans, 29
- uses for financial fraud, 63–64
- viruses, 23
  - origin of term, 46
  - theory, 46
- worms, 27
  - Blaster, 90
  - Welchia, 89, 92

malware
- definition of, 14

Mersenne Twister, 110
Metamorphism, 43
Metasploit, 64
MinGW, 54

Naive Bayes Classifiers, 68–69
Neural Networks, 48, 69–70
NtSystemQueryInformation(), 101

Online Banking security, 55–65

Payload, 35
Police and Justice Act 2006, 44
Polymorphism, 43
Portable Executable (PE) File Format, 78
Predictive Modelling, 66
Privilege Escalation, 94
Process Network Usage, 104

Process Trees, 104
PROCESS_MEMORY_COUNTERS, 101
Processes in Windows, 93
PSAPI, 100

Real-time Data Collection, 123
Registry, 94
Relocatable Code Tricks, 84
Resource Hogging, 97
    due to bad programming, 97
    due to deliberate strategy, 97
Retroviruses, 44
Rootkit, 35, 103
RVA, 108

Sampling process, 123
Sasser worm, 27
Section Table (PE), 79
Self-Modification, 91
Shellcode, 84
Signature Scanners, 40
Simulators, 98
Spam, 34
Spyware, 30
    identification criteria, 31
Stack Overflow, 36
Static Analysis, 49
Structural Anomalies, 80
Structural Classification, 106–120
Structure-Based Detection, 41

Trojan, 29, 46
Trusted Computing, 51
    Palladium/NGSCB, 51

UPX, 91, 109

Virtualisation, 135
Virtualisers, 99
Virus, 23, 46, 47, 77, 80
    appending and prepending, 24
    boot sector, 25
    characteristic structural features, 77
    definition, 23
    macro, 26
    memory-resident, 24
    Michaelangelo, 25
    overwriting, 24
VMWare, 99
Vulnerabilities, 36
    code injection, 36
        buffer overflow, 36
        shellcode, 37

Wardriving, 16
WEKA, 73–74, 107
    ARFF format, 73
    cross-validation
        definition of output stats, 159
    Example Data Files, 73
Windows Performance Data, 100
Windows Vista, 52
Wireless Networks, 16
Worm, 27
    Blaster, 27
    email, 27
        Love Bug, 28
    Internet Worm, 27
    Sasser, 27
    Xerox experiments, 46

Xen, 99

Zombies, 32