MISSOURI S&T CURTIS LAWS WILSON LIBRARY MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Scholars' Mine

Masters Theses

Student Research & Creative Works

Fall 2011

Adaptive rule-based malware detection employing learning classifier systems

Jonathan Joseph Blount

Follow this and additional works at: http://scholarsmine.mst.edu/masters_theses

Part of the Computer Sciences Commons

Department:

Recommended Citation

Blount, Jonathan Joseph, "Adaptive rule-based malware detection employing learning classifier systems" (2011). *Masters Theses*. 5008.
http://scholarsmine.mst.edu/masters_theses/5008

This Thesis - Open Access is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Masters Theses by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

ADAPTIVE RULE-BASED MALWARE DETECTION EMPLOYING LEARNING CLASSIFIER SYSTEMS

by

JONATHAN JOSEPH BLOUNT

A THESIS

Presented to the Faculty of the Graduate School of

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2011

Approved by

Daniel R. Tauritz, Advisor
Bruce M. McMillin
Samuel A. Mulder

Copyright © 2011
Jonathan Joseph Blount
All Rights Reserved

ABSTRACT

Efficient and accurate malware detection is increasingly becoming a necessity for society to operate. Existing malware detection systems have excellent performance in identifying known malware for which signatures are available, but poor performance in anomaly detection for zero day exploits for which signatures have not yet been made available or targeted attacks against a specific entity. The primary goal of this thesis is to provide evidence for the potential of learning classifier systems to improve the accuracy of malware detection.

A customized system based on a state-of-the-art learning classifier system is presented for adaptive rule-based malware detection, which combines a rule-based expert system with evolutionary algorithm based reinforcement learning, thus creating a self-training adaptive malware detection system which dynamically evolves detection rules.

This system is analyzed on a benchmark of malicious and non-malicious files. Experimental results show that the system can outperform C4.5, a well-known non-adaptive machine learning algorithm, under certain conditions. The results demonstrate the system's ability to learn effective rules from repeated presentations of a tagged training set and show the degree of generalization achieved on an independent test set.

This thesis is an extension and expansion of the work published in the Security, Trust, and Privacy for Software Applications workshop in COMPSAC 2011 - the 35th Annual IEEE Signature Conference on Computer Software and Applications [1].

ACKNOWLEDGMENT

I would like to express my gratitude to the late Dr. Ann Miller for providing support during her time as my advisor. She always encouraged me to pursue what I found interesting. Her advice helped guide me towards the field of computer security and her dedication to learning will be remembered.

I thank my advisor Dr. Daniel Tauritz for his help, suggestions, technical guidance, and keeping me motivated throughout my career as a graduate student. His dedication to his students helped me stay focused and enabled me to complete this thesis. As an undergraduate, his courses were stimulating and became foundations for my graduate work. His role as my advisor was paramount in helping me secure research and career opportunities.

I would like to thank Dr. Samuel Mulder of Sandia National Laboratories for his support and feedback throughout this thesis. His expertise and advice helped guide me and his insights kept me headed in the right direction.

Dr. Bruce McMillin deserves my thanks as well. I had the privilege of taking his courses as an undergraduate as well as a graduate student and appreciate the knowledge I have gained from him.

I would also like to thank Danny Quist of Offensive Computing for providing the malware samples used in this research.

Additionally, I would like to thank everyone at Sandia National Laboratories who have supported my education and helped fund it. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Finally, I would like to thank my friends and family for their support throughout my education.

TABLE OF CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGMENT iv

LIST OF ILLUSTRATIONS viii

LIST OF TABLES ix

SECTION

1. INTRODUCTION 1
2. MALICIOUS SOFTWARE 3
   - 2.1. COMMON TYPES OF MALWARE 3
   - 2.2. MALWARE DETECTION 5
     - 2.2.1. Anomaly-based detection 6
     - 2.2.2. Specification-based detection 7
     - 2.2.3. Signature-based detection 7
3. RELATED WORK 9
4. BENCHMARK DESIGN 12
   - 4.1. DATASET 12
   - 4.2. PRE-PROCESSING 13
     - 4.2.1. VirusTotal 13
     - 4.2.2. Feature Extraction 14
     - 4.2.3. Executable Packing 16
     - 4.2.4. Entropy 17
   - 4.3. FILTERING THE DATASET 21
   - 4.4. C4.5 DECISION TREE ALGORITHM 22
   - 4.5. C4.5 BENCHMARKING RESULTS 22
5. EVOLUTIONARY COMPUTING 26
   - 5.1. EVOLUTIONARY ALGORITHM COMPONENTS 27
     - 5.1.1. Representation 27
     - 5.1.2. Fitness function 27
     - 5.1.3. Population 28
     - 5.1.4. Parent selection 28
     - 5.1.5. Recombination 28

5.1.6. Mutation 28
5.1.7. Survivor selection 28
5.1.8. Initialization 28
5.1.9. Termination 28
5.2. GENETIC PROGRAMMING 28
5.2.1. Representation 29
5.2.2. Initialization 29
5.2.3. Recombination 29
5.2.4. Mutation 30
6. LEARNING CLASSIFIER SYSTEMS 31
6.1. DISCOVERY 32
6.2. LEARNING 32
6.3. MICHIGAN AND PITTSBURGH STYLE LCS 33
6.4. OVERGENERAL CLASSIFIER PROBLEM 33
6.5. STRENGTH-BASED AND ACCURACY-BASED FITNESS 34
6.6. ZCS 34
7. EXTENDED CLASSIFIER SYSTEM (XCS) 37
7.1. INITIALIZATION 39
7.2. EVALUATION 39
7.3. EVOLUTION 41
8. SYSTEM DESIGN 42
8.1. LEARNING CLASSIFIER SYSTEM 42
8.1.1. Rule Representation 42
8.1.2. Initialization 42
8.1.3. Action Selection 44
8.1.4. Rule Evaluation 44
8.1.5. Mutation 45
8.1.6. Crossover 45
8.1.7. Evolutionary Algorithm 45
8.2. DECISION TREE INITIALIZATION 46
8.2.1. C4.5 rule initialization 46
8.2.2. Specialization function 47
8.3. PERFORMANCE METRICS 47
8.4. DEFAULT PARAMETERS 48
9. EXPERIMENTAL RESULTS 51
9.1. FAMILY OF MALWARE STUDY 51

9.2. BENCHMARK DATASET COMPARISON 56
9.2.1. Initialization method study 56
9.2.2. Population size study 58
9.2.3. Offspring size study 58
9.3. SYSTEM COMPARISON TO C4.5 DECISION TREE 59
10. CONCLUSION 65
11. FUTURE WORK 66
BIBLIOGRAPHY 68
VITA 72

LIST OF ILLUSTRATIONS

Figure Page

4.1 Pre-processing stage 13

4.2 Sample output from VirusTotal 14

4.3 Number of imports per file 15

4.4 Most common imports 16

4.5 Number of PE sections 19

4.6 Number of high-entropy PE sections 20

4.7 Portion of a decision tree created using C4.5 23

4.8 C4.5 Results 24

5.1 Evolution Cycle 27

5.2 Genetic programming tree representation 29

6.1 ZCS overview 36

7.1 XCS overview 38

7.2 XCS rule representation 38

8.1 Malware LCS Diagram 43

8.2 Visualization of a subtree of a generated rule 43

9.1 Malware family results using population size=10 52

9.2 Malware family results using population size=20 53

9.3 Malware family results using population size=30 54

9.4 Population size training results 59

9.5 Population size testing results 60

9.6 Visualization of an unbalanced C4.5 decision tree 62

9.7 C4.5 and LCS comparison 63

9.8 LCS evolution compared to C4.5 64

LIST OF TABLES

Table Page

4.1 PE file errors preventing feature extraction 20

8.1 Classifier system parameters 49

9.1 Family of malware study experimental results 56

9.2 Initialization study experimental results 57

9.3 Offspring size study experimental results 61

1. INTRODUCTION

Malicious software (malware) is a program that causes undesired behavior or damage to a computer system, network, or service. Undesired behavior of software includes disruption or denial of service, loss of privacy, unauthorized access to resources, and any other hostile or intrusive behavior. Malicious software runs without the consent of users and often without their knowledge. Malware is an ever-growing threat to computer systems, and security researchers are in a virtual arms race with malware authors. The number of different strains and types of malicious software has been on the rise for years. A recent report states that "in 2010, cyber-criminals created and distributed a third of all existing viruses", 34% of all malware that has ever existed until 2010 was created in those 12 months [2].

Trends in computer usage have led malware authors to create malicious software on any platform that is profitable. Cybercrime is a multi-million dollar business [3], and will continue to get larger as more computing devices become available and are able to get online. As with any other business, internet crime is driven by a return on investment, and new computing areas will offer criminals new venues of profit. Once a computer platform has enough users to be worth targeting, criminals will start attacking that system. Mobile threats are a new category for malware, and as the smartphone user base grows, so will the malware author base targeting mobile platforms. In 2010, 163 mobile vulnerabilities were reported [4] and some of the first trojans and mobile botnets started infecting phones. Attackers are actively seeking out new areas that can be exploited for profit.

This thesis applies learning classifier systems (LCSs) to malicious software detection. A learning classifier system is a rule-based system that utilizes reinforcement learning combined with evolutionary computing to model an intelligent decision maker in an arbitrary environment. LCSs are based on Darwinian theories and their biological-inspired techniques are what makes them adaptable to many environments and problems.

This thesis presents a blueprint for a benchmarking system for malware detection, using a dataset of malicious and non-malicious files. It describes a feature

extraction stage of malicious executables as well as introduces a method for training and testing adaptive learning classifier systems on a dataset representative of a real computer system. This thesis also contributes a baseline performance analysis of the C4.5 machine learning algorithm on the dataset for comparison with other techniques. Most importantly, this work introduces a hybridization of XCS and S-classifiers for malware detection, adapting it for maximal accuracy and generalization as shown on the benchmarking dataset.

2. MALICIOUS SOFTWARE

The classification of malware is a difficult problem. Software that allows unauthorized control of a system is obviously malicious, but software that displays ads (adware) is not strictly harmful, unless it invades privacy and collects personal information without user consent (spyware). Software can be considered malicious depending on the intent of its author, and this makes it difficult to classify a piece of code as malicious or not. Research has automated malware detection based on the intent of the user and the intent of the malware author [5].

Malware comes in various forms and categories, and are classified according to the methods and techniques employed by the software. While there is overlap between categories and most malware falls into multiple categories, making a clear distinction between the different types of malware is useful so the reader has background knowledge on the different kinds of threats detection software must identify.

2.1. COMMON TYPES OF MALWARE

The following list presents common types of malware paraphrasing existing literature [6, 4, 7, 8].

- Virus - Computer viruses propagate from one file to another and/or from one computer to another by inserting their code into other programs. Viruses need an existing host program to execute and cause harm.

- Worm - A worm is a type of virus that spreads to many computer by copying itself to another computer on the network. Worms typically infect as many connected systems as possible and unlike viruses, do not need a host in order to cause damage.

- Trojan horse - Trojans mask themselves by appearing to be something legitimate. Trojans typically destroy data or attempt to extract confidential information including financial data and passwords.

- Spyware - Spyware tracks users' surfing habits and online information in order to display targeted ads. Spyware tries to ensure it remains on a system and is considered an invasion of privacy.
- Rootkits - Rootkits hide traces of themselves from the computer and the user to evade detection in order to run longer and increase the damage dealt to a system.
- Scareware - Rogue security software that pretends to be legitimate and creates false alerts that prompt users to download or purchase malicious software, sometimes installing the malware it claims to protect against
- Backdoor - A backdoor bypasses normal authentication and security mechanisms to allow access and control of a system.
- Keylogger - A program that stores strokes typed on a keyboard, allowing attackers access to confidential data including passwords and financial data.

In the last few years, malware authors have developed advanced techniques of spreading their malicious creations. The following list of newer techniques is paraphrased from existing literature [4, 6, 8].

- Phishing - Sending email that falsely claims to be from a legitimate source in order to steal personal information.
- Spear Phishing - A type of phishing that targets a group of people that have something in common, e.g., working at a company or going to the same school, and utilizes inside information to appear legitimate and from a trusted source
- Targeted attacks - These high profile attacks are typically directed at enterprises and governmental organizations and are a form of spear phishing that target specific individuals or groups with privileged access
- Attack kits - These software toolkits are sold by malware creators to common cyber criminals for widespread use. As opposed to targeted attacks, these packages infect a website and use multiple attack vectors that work cross-browser on multiple platforms to exploit as many visitors as possible.

2.2. MALWARE DETECTION

Malware detection is the primary step in preventing a computer system from potential information loss and system compromise. There are a variety of ways to detect malicious software. A malware detector attempts to detect malicious behavior or programs. Detection is the most important step in preventing a computer system from being infected, protecting it from potential information loss and system compromise. There are many methods of detecting malicious software to prevent it from executing. This section details some of the common techniques and approaches and their advantages and disadvantages.

The main detection techniques are anomaly-based, signature-based, and specification based detection. These techniques are not limited to malware detection, many of these are applicable to network intrusion detection, fraud detection, and other fields.

A majority of anti-virus software use signature-based techniques that utilize a continuously updated set of signatures [9]. This is a reactive technique: until a signature is created for an exploit, the exploit will elude detection by traditional anti-virus software.

Anomaly-based techniques detect patterns in data that differ from expected behavior. Anomalies in data can correspond to important changes in a system. A computer with an anomalous internet traffic pattern could indicate it has been attacked and is disclosing information [10].

The approach of a malware detection system determines how the information about a program is gathered and the detection technique determines how that information is used. The main approaches to malware detection are static analysis, dynamic analysis, and a hybrid of the two.

Static analysis uses structural and syntactical information of a program to determine its maliciousness. Before a program is executed, static information found in the executable, including header data and the sequences of bytes, is used to determine whether it is malicious. Static analysis is able to analyze all possible paths a program may take at run time, but usually not every possible state of the program as there are an arbitrary number of user inputs and states [11]. Static analysis uses an abstracted model of program that approximates the actual behavior [11]. Static analysis can be time consuming, especially with obfuscated code.

Dynamic analysis uses run-time information including contents of the run-time stack to detect malware during or after its execution [8]. Dynamic analysis does not use abstraction, it analyses actual run-time behavior of a program. Dynamic analysis analyses a single execution path of an executable and multiple runs using different sets of inputs will cover additional code paths. There is no guarantee, however, that all possible behaviors of an executable are covered by a set of inputs. The main challenge of static analysis is creating a good abstraction of the program and the primary challenge of dynamic analysis is creating a representative set of inputs to the program [11].

2.2.1. Anomaly-based detection. Anomaly-based detection techniques compare a program's behavior to what is considered normal behavior in order to evaluate whether or not a program is malicious. This technique has two phases, a learning, or training, phase and a detection, or monitoring, phase. During the learning phase, a profile of normal and acceptable behavior is learned by the detection system. During the detection phase, this learned profile is applied to actual activity in order to detect deviations (anomalies).

A major advantage of an anomaly-based technique is the ability to detect zero-day attacks which the system has never seen before. The disadvantages of anomaly-based detection techniques are a high false alarm rate and the difficulty of determining what type of features the system should learn during the training phase [8]. A primary challenge in detecting anomalies is defining acceptable behavior, which includes every possible normal behavior, and creates a precise boundary between normal and anomalous activity. Behavior that lies close to this boundary has a higher chance of being incorrectly classified.

Dynamic anomaly-based detection uses information gathered during a program's execution. The detection phase identifies malicious behavior by comparing it with learned behavior. Dynamic techniques execute malicious programs on the system.

Static anomaly-based detection looks at the file structure of the program and uses its characteristics to determine if it contains malicious code. An advantage of static anomaly-based detection is that malware can be detected without having to execute it [8].

Rule-based anomaly detection techniques learn rules during the training phase that capture the normal behavior of a system. Any behavior that is not covered by

the rules is considered to be an anomaly. Association rule mining-based techniques have been used for network intrusion detection, system call intrusion detection, and credit card fraud detection [10].

In an ideal situation, the set of anomalous activities would be the same set as malicious activities. If this were the case, computer security would be an easy problem to solve; however, in reality systems encounter false negatives, behavior that is not anomalous but is malicious, and false positives, behavior that is anomalous but not malicious. While training anomaly thresholds can be set low to minimize false negatives, this will increase the amount of false positives. The training profile is only an approximation of all valid behaviors and valid behaviors not seen during the training would cause false negatives [8].

2.2.2. Specification-based detection. Specification-based techniques are similar to anomaly-based techniques, but differ in the source of the acceptable behavior. While anomaly-based systems typically develop normal profiles of systems using automatic training, in specification-based detection, security specifications are manually created for correct behavior of critical objects. Programs that violate these specifications are anomalous, and unknown attacks can be detected.

Specification detection differs from anomaly detection by approximating the requirements of a system instead of approximating an implementation of one. Training a system specifies all valid behaviors of that system. A main challenge of specification-based techniques is completely and accurately specifying the entire set of acceptable behaviors, as any inaccuracies result in incorrect detection.

An example hybrid specification approach combines static with dynamic analysis to analyze system calls [12]. During static analysis, information is saved about every system call which includes its 1) call address, 2) name, and 3) return address. During dynamic analysis, executables are monitored during run-time to ensure each system call matches the list from static analysis. If a process was modified during run-time, e.g., it was injected by a malicious process, then the three pieces of information will have changed and the behavior will be detected.

2.2.3. Signature-based detection. In signature-based detection schemes, the system attempts to model the malicious behavior of programs and compares each program to the signatures. The collection of signatures represents all the knowledge

the system has about how malicious programs behave and this collection is used to make a decision on the maliciousness of a program. For a program to be detected as malicious it must match a signature known by the detection system. Signatures must be updated when a new piece of malware is found; the system has to be constantly checked to ensure it is up-to-date. This creates a problem when a zero-day attack emerges, as it takes time to create and push out a signature and the system is vulnerable during this time. The quicker signatures can be created, the better they can protect against malware that spreads quickly throughout the internet. There has been some research into automated signature builders, but they are not as advanced as humans [8].

The primary drawback of signature-based detection is the set of possible malicious behaviors is infinitely large, and there are no known techniques that can accurately represent the entire set. As new malicious behaviors are found, signatures have to be created in order to stay up-to-date. New malware is often derived from existing malware, and signatures can take advantage of this fact if they are created in such a way that captures the malicious essence of the malware; benefits of this would include that a smaller number of signatures would be as effective, and the ability to detect obfuscated versions of the same malware.

3. RELATED WORK

Recent research has shown that existing anti-virus software is poor at identifying polymorphic scripts [13]. By using various polymorphic techniques, the authors were able to create malicious scripts with identical functionality to known malware that were undetectable by existing anti-virus software. They were able to detect the malware variants by analyzing the software's dependency graph with a hybrid genetic algorithm. Simple polymorphism could fool more than half of the software that detected the original malware.

There have been approaches to malware detection that use non-signature based techniques. By extracting features of portable executable (PE) files, malicious executables can be detected by heuristic techniques [9]. Packed and non-packed files were processed separately by a decision tree to determine whether they are maliciousness. To determine whether a binary is packed, three features were looked at: (1) the name, number and type of sections, (2) the number of entries in the import address table, and (3) the entropies of various portions of an executable. Different structural models were developed for packed and non-packed executables. This technique was shown to overcome a bias shown by structural features for packed/non-packed executables by using two models: the non-packed model uses a subset of features for non-packed files which contain the most information, and the packed model uses a subset of features which are least perturbed by packing.

A comparative study analyzed evolutionary and non-evolutionary rule learning algorithms to evaluate performance differences [14]. Five types of LCS were compared with five non-evolutionary rule learners. Four performance metrics were used to compare the algorithms: (1) classification accuracy, (2) the number of rules, (3) comprehensibility of the rules, and (4) processing overhead. All algorithm implementations were provided by a unified framework called Knowledge Extraction based on Evolutionary Learning (KEEL). The rules were built from 189 features extracted from the files, but examples of features were not provided and an unknown number of features were determined to be redundant. Each algorithm was run on a subset of 10,339 malicious executables. The total malicious set of executables was divided into cate-

gories such as backdoor, virus, or trojan. The size of each subset ranged from a few hundred files to 2500, with the average below 1500 executables. The ratio of training set size to test set size was 9:1, so each category has a test set of only a few hundred files. The reported classification accuracy for all algorithms was above 95 percent. Their conclusions indicated that the non-evolutionary rule learning algorithms outperformed the LCS types. The LCS types still had very high detection rates, but at the expense of high processing time. The authors acknowledged that they did not explore different configuration parameters for the rule learning algorithms, as well as a plan to combine the datasets into a single large set in order to create a more challenging environment. Limiting the sets to a specific category of malware increases the chances of similarity between files, providing for an easier environment than a random collection of malware; this narrow result led to artificially high accuracy rates which are most likely not representative of real world performance.

A fuzzy learning classifier system was applied to intrusion detection in [15]. The system automatically generated a detection model that could detect known attacks and variations on known attacks by using a fuzzy learning classifier system. Results were presented on a benchmark dataset and compared with the C4.5 classification algorithm. The system evolved fuzzy classification rules for effective intrusion detection by optimizing membership functions using an evolutionary algorithm. The fuzzy rules were constructed based on the C4.5 rules and compared to the standard C4.5 approach. Each individual in the population consisted of membership functions related to input variables from a dataset of intrusions. Four different input variables were used in the individual representation: duration, source bytes, destination bytes and a measurement of the number of "hot" indicators. These input variables contain a total of 66 real-valued attributes that were fuzzified for use in the fuzzy learning classifier system. Two separate experiments were carried out, the first constructed fuzzy rules based on the C4.5 rules. Fuzzy rules were created using a genetic algorithm and had a significant improvement over the original rules. An implementation of a fuzzy learning classifier was then applied to the dataset and resulted in a slight improvement of the rules.

A Genetically Programmed Learning Classifier System was used in [16] and was originally designed to be an intelligent agent used to support the maintenance of the

Joint Strike Fighter aircraft. The system was able to learn to optimize its results of tasks including inventory purchase, and mobile agent functionality. The system used genetic programing for representation and used a full bucket-brigade fitness passing approach for learning and reinforcement. Crowding helps maintain diversity of the population by comparing children to similar adults and replacing the least fit adult with the new child. The system design was similar to the classic learning classifier system as it uses an internal message list.

4. BENCHMARK DESIGN

4.1. DATASET

The dataset for this research consisted of malicious executables as well as non-malicious ones. Malicious software samples were obtained from Offensive Computing [17]. Non-malicious software samples were obtained from a machine freshly installed with Windows XP and a university campus computer learning center machine running Windows XP. Clean executables included software created by Microsoft, Adobe, MathWorks, other third parties, and open source software.

A primary assumption this thesis makes is that malicious files will be distinguishable from goodware based upon the structure of the PE file including section names and entropies as well as the import address table. While in the real world this assumption would not always hold, for the purpose of this research, this is an acceptable assumption for determining whether an LCS can potentially be used for malware detection.

Windows executables, both malicious and non-malicious, are written in the portable executable (PE) format. PE files start with a DOS header, followed by a PE header, and contain a number of sections. Each section has a header which describes its data and resources. A typical Windows application has nine predefined sections: .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug [18]. Each section has a different purpose, for example .text is used to store program code while .data is used for global variables.

One important section is the import data section which contains the Import Address Table (IAT). The IAT is where every external function called by an executable is stored. This table includes the name of the function and the name of the dynamic link library (DLL) in which the function is stored.

Different types of files contain a variety of imports, and the number of imports a file includes depends on its purpose. For example, kernel32.dll is a common dynamic link library which provides functionality to all of the fundamental Windows application programming interfaces. This includes access to memory management,

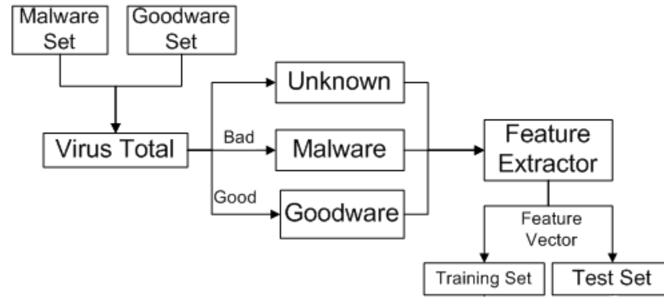


Figure 4.1: Pre-processing stage

file system input/output, process and thread creation, and error handling. Many user programs import functions from kernel32.dll to provide basic functionality. kernel32.dll itself imports these functions from the native API DLL ntdll.dll, and has 391 imported functions. This high of a number of imports is uncommon for most executable files, but is not so uncommon for DLLs.

4.2. PRE-PROCESSING

An overview of the pre-processing method is shown in Figure 4.1. This stage takes each file from the dataset and passes it through VirusTotal to determine its maliciousness, and then extracts a set of features from the file.

4.2.1. VirusTotal. VirusTotal [19] is a online web site that offers a free analysis of files. It is an independent service that runs multiple anti-virus engines on each file to identify viruses, worms, trojans and other malicious software. VirusTotal maintains an updated version of each anti-virus engine to ensure they include the latest virus signatures. At the time of this writing, 43 different anti-virus engines were in use.

Each executable file used in this research was submitted through VirusTotal's API for analysis. The result from VirusTotal was used as the ground truth in deciding whether or a not a file was considered malicious or not.

The set of samples from Offensive Computing does not contain just malware. If more than 25 percent of the anti-virus vendors VirusTotal uses to scan a file report malicious, then the file is considered malicious. This eliminates the possibility of a single anti-virus software misclassifying a file. Each file was checked to make sure that

File name: 70a13374
Submission date: 2009-10-03 10:30:14 (UTC)
Current status: finished
Result: 34 /41 (82.9%)

Compact

Antivirus	Version	Last Update	Result
a-squared	4.5.0.24	2009.10.03	Trojan-Spy.Banker!IK
AhnLab-V3	5.0.0.2	2009.10.02	-
AntiVir	7.9.1.27	2009.10.02	TR/Dropper.Gen
Antiy-AVL	2.0.3.7	2009.10.03	-
Authentium	5.1.2.4	2009.10.02	W32/SuspPack.M.gen!Eldorado
Avast	4.8.1351.0	2009.10.02	Win32:Spyware-gen
AVG	8.5.0.420	2009.10.03	SHeur2.AECQ
BitDefender	7.2	2009.10.03	Trojan.Spy.Bancos.NLA
CAT-QuickHeal	10.00	2009.10.03	Win32.TrojanDownloader.Banload.1
ClamAV	0.94.1	2009.10.03	-

Figure 4.2: Sample output from VirusTotal

VirusTotal is consistent with the dataset the file came from; i.e., a sample from the malware set must be identified as malicious and a sample from the goodware set as non-malicious. If it is not consistent, the sample is considered unknown. Furthermore, all samples (malicious or not) that are not in VirusTotal's database are also considered unknown. All unknown samples are not used in the results presented in this thesis, as they cannot be verified as being definitely malware or goodware.

An example of a file submitted through VirusTotal's web interface is shown in Figure 4.2. The results from a selection of the anti-virus engines are shown for one of the pieces of malware used in this research. During preprocessing, if these results indicate a file is malicious, it is marked as such in the dataset.

4.2.2. Feature Extraction. Feature extraction takes features and characteristics of the executables and turns the information into a format used by the LCS. The IAT from each executable is parsed to generate a feature list containing all of the imported functions each executable references. The list of sections is also extracted, along with their sizes and entropies. The feature extractor implemented using an open source tool called pefile [20].

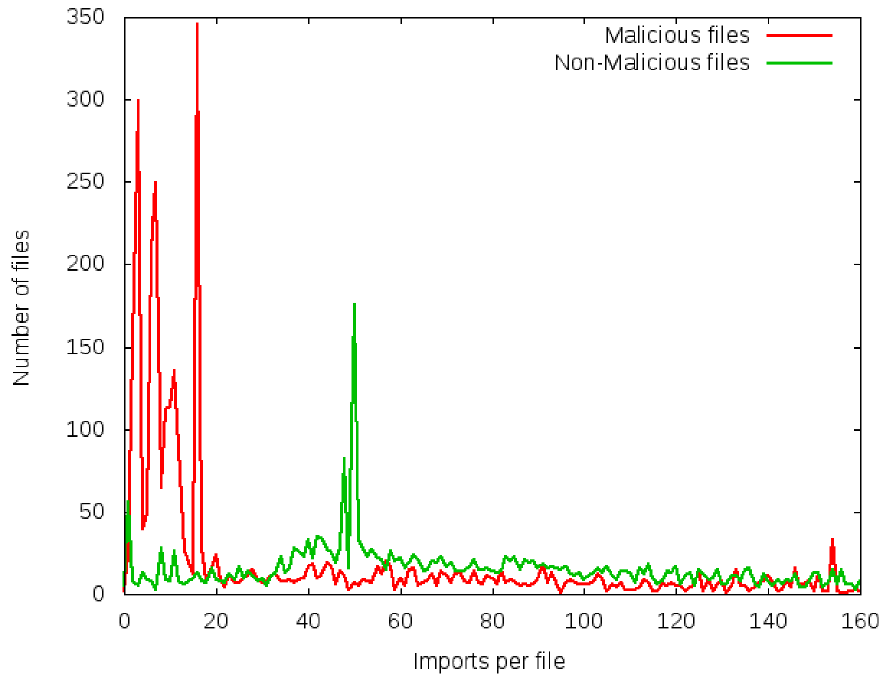


Figure 4.3: Number of imports per file

Details of the full dataset and extracted features are as follows:

- 6,774 total files
- 3,401 malicious files
- 3,373 non-malicious files from Windows XP
- 58,584 total unique imports
- 703 total unique sections

A graph of the number of imports per malicious and non-malicious file is shown in Figure 4.3. An interesting note is how many malicious files have a low number of imports, and the most common number of imports for clean files is much higher than that of malicious files.

Looking at the most common imported functions, Figure 4.4 shows the number of malicious and number of clean files that imported each function. All but one of these functions exist in kernel32.dll, the fourth most popular came from advapi32.dll

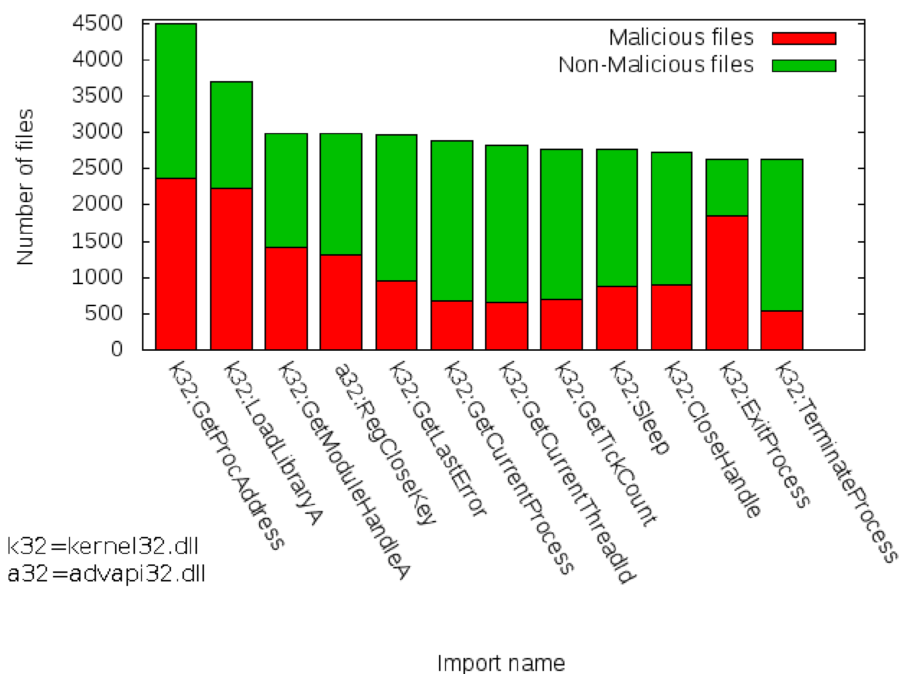


Figure 4.4: Most common imports

which provides access to advanced services of Windows, including the registry. The most popular imports were split about 50-50 in the number of goodware and malware files that use those functions. One outlier was $kernel32.dll.ExitProcess()$, this function was imported more by malicious files than clean ones.

4.2.3. Executable Packing. Executable packing is a form of compression and/or encryption of executable files, and the programs that perform the compression are known as packers. Compression programs are utilized to minimize disk space usage as well as network transfer times. Traditional programs like WinZip compress files into a separate archive which must be extracted to disk to retrieve the original files. Packers compress an original program into a new executable and add a stub function which will unpack (decompresses and decrypt) the original executable into memory. A packed executable runs without having to be separately extracted, unlike .zip files. This process obscures the original file and makes it more difficult to reverse engineer. One of the most common packers is the Ultimate Packer for eXecutables (UPX). UPX is a free and open source executable compression tool that can be used on Windows binaries as well as other platforms. UPX by itself is not malicious, but many types of files including malware are packed using it.

When malware is packed, the size of the file is decreased, but more importantly the signature of the file changes and this makes it more difficult for anti-virus software to detect. Two potential downsides to packing are the additional computation overhead at run-time to unpack the original and the fact that the packing process creates a signature which can be detected [21]. All packers have the same weakness: they must unpack the original executable into memory in order to execute it; they can not stop analysis of a file, they can only make it more costly [21].

Around 90%, of malware samples use some form of packing [22]. Popular packers (ASpack, FSG, Morphine, UPX, MEW) are detected by most anti-virus software, but many of the less common ones have very poor detection rates [22]. By using multiple packers on the same family of malware, malware authors can evade detection for longer periods of time. Some scanners incorrectly flag packed but legitimate executables as malicious or suspicious. There are advanced packers that make analysis harder, by including routines to make memory debugging and dumping more difficult, and not all anti-virus products account for the various types of packers.

4.2.4. Entropy. Entropy in information theory is a measure of uncertainty in a series of bytes [23]. Similar to how sentences are not random sequence of letters, Windows executables are not random sequence of bytes, and the entropy of a block of data is a measurement of uncertainty in those bytes. Having a high entropy does not imply randomness; compression creates a high level of entropy but leaves data highly structured [24]. Compressed and encrypted data will look very similar from an entropy point of view.

In binary files, entropy can be calculated by counting the number of times each byte occurs. Entropy of a discrete random variable x can be calculated by the formula [23]:

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{1}$$

where p is the probability mass function of x and $p(x_i)$ is the probability of the i_{th} unit of information. One byte contains 8 bits and there are $2^8 = 256$ possible values (00h - FFh). The unit of entropy in Equation 1 is the bit, and entropy ranges from $0.0 - 8.0$ for a sequence of bytes.

Analysis of different types of executables, including native windows executables, packed executables, and encrypted executables, resulted in different levels of entropy for each file type [24]. The results show that executables with an average entropy of 6.677 or greater are statistically likely to be packed or encrypted. Malware authors can conceal encryption or compression by adding in redundant bytes or including invalid blocks or blocks with all zero values to lower a binary's entropy. Malware executables that exceed this average entropy are considered to be packed.

Each PE file contains a number of sections that the operating system loads, some contain data and some contain executable code. Malware can make analysis more difficult by using nonstandard sections and renaming them. All of the malware samples in this research were analyzed in a pre-processing stage. During this analysis, each section of each file was analyzed for its name, size, and entropy. Figure 4.5 shows the top 11 PE sections in the data set, shown by the total number of sections used in malicious and non-malicious files. Eight of the sections are generated by standard compilers and linkers and they are: .rsrc, .data, .text, .reloc, .rdata, .idata, .bss, and .edata. Of the remaining three sections, UPX0, UPX1 are generated by the UPX packer and the third section's name contained a string of the unprintable byte '\x00' and is not a standard name section. Malware may create a non-ASCII or random name of a section to obfuscate its purpose.

From the set of clean files, 108 unique section names were found, and 622 different section names were found in the set of malicious files. The large difference is due to malware creating non-standard names to make analysis more difficult. Sections that have a size 0 are uninitialized initially and when the program is run and copied into memory, these 0-byte sections could get filled with data or unpacked code. Ignoring empty sections, 26.4% of all the malicious sections in the dataset were determined to be packed; and 2.4% of the non-malicious sections. Each file contains multiple sections, and 69.2% of malicious files contained at least one section that was determined to be compressed or encrypted, while 10.2% of non-malicious files had one or more packed sections. Malicious programmers are more likely to pack their executables, as it saves network transfer time and disk space and some packers make it more difficult for anti-virus software to detect and reverse engineer.

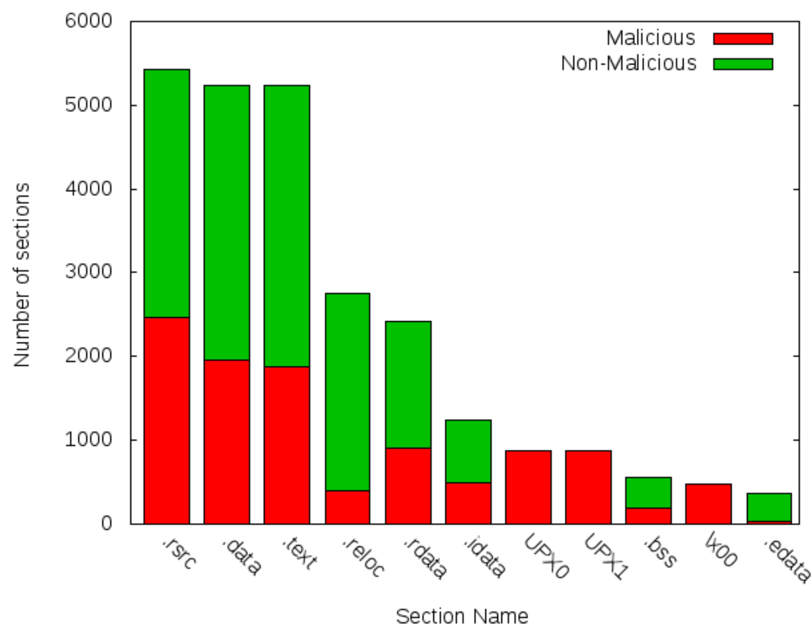


Figure 4.5: Number of PE sections

Different packers compress their data into different sections, some use the default ones, and others create their own sections (as shown by UPX). Figure 4.6 shows the most popular sections that contained data with high entropy and are most likely packed. The most popular packed section was UPX1, which is the section UPX use to store the compressed data of the original executable. A small amount of non-malicious files had packed UPX1 sections. As mentioned earlier, UPX is not only used maliciously, it also has legitimate uses. Seven clean third-party executables were packed using UPX including gpg.exe - Gnu Privacy Guard, a file and email encryption tool and winscp.exe - an open source SFTP and FTP client for Windows. There are legitimate programs that use packing tools, but they are in the minority, and no Windows files were found to be packed.

The other top packed sections include default ones, the string of '\x00', and UPX2. Interestingly, the only packed section that more non-malicious files contained than malicious was .reloc. The .reloc section holds a table of base relocations, which are adjustments to an instruction if the loader cannot find a file. The relocation data in this section inherently has a high entropy, and is not necessarily the product of a packer. The UPX0 section seen in Figure 4.5 is the section that UPX uses to unpack

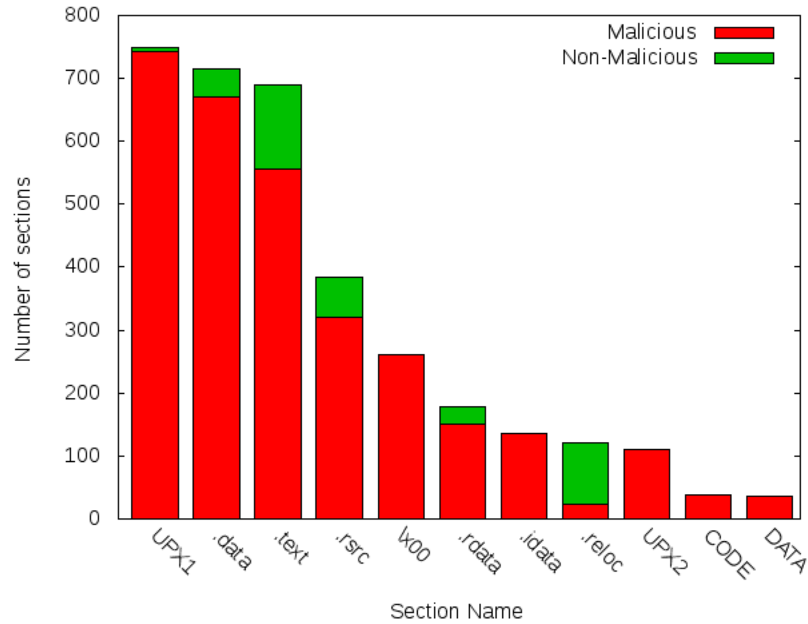


Figure 4.6: Number of high-entropy PE sections

Table 4.1: PE file errors preventing feature extraction

pe instance has no attribute 'directory_entry_import'
invalid nt headers signature.
no optional header found, invalid pe32 or pe32+ file
data at rva can't be fetched. corrupt header?
data length less than expected header length.
invalid e_lfanew value, probably not a pe file

the original executable into memory, it initially starts empty on disk and therefore has a entropy of 0, so it does not appear in Figure 4.6.

Files with corrupt or missing IATs could not be analyzed in this system, Table 4.1 lists all encountered PE errors reported by pefile. 847 files were found to be incompatible for feature extraction. These files could be malicious executables that have been purposely modified by their authors to change the content of their PE header in order to make analysis harder.

4.3. FILTERING THE DATASET

Occasionally, a malicious file has an identical set of features as a non-malicious file. Any detection system would inherently not be able to tell the difference between files with identical features. These files were removed from the dataset to increase the usefulness of the evolved rules. Files with corrupt or missing IATs could not be analyzed in this system. Extracting additional features would improve the number of files that could be used in the system; these features are discussed in the future work section.

As described earlier, many malicious programs are packed, and building a system to identify packed files is trivial as they have a much different structure than unpacked files. A detector that can identify packed files may do reasonably well as a malware detector, considering many malware samples are packed, and not many goodware files are packed. It is also easy to separate .dll files from regular .exe executables, and although link libraries are written in the same format, they are not directly executable themselves but provide functionality to other programs. The dataset was hardened by removing packed executables, as well as clean files such as link libraries that were not .exes. This subset of files is more difficult to distinguish, as the features have more in common. The details of the unpacked and filtered dataset are as follows:

- 854 malicious files
- 1,048 non-malicious files
- 18,804 unique imports
- 126 unique sections

There are two types of noise that classification and data mining problems can have [25]. The first type, known as classification noise, makes it impossible for the system to develop to achieve 100% testing accuracy; the predictive attributes of the training set do not allow perfect prediction of the test set. The second type of noise, attribute noise, are attributes within the problem domain that aren't useful in the system's prediction, they have no relationship to class. These attributes are known as non-predictive attributes. In certain problems the distinction between predictive

and non-predictive attributes is a goal of the system, and noise in the dataset is a common and important problem [25].

4.4. C4.5 DECISION TREE ALGORITHM

C4.5 is a machine learning decision tree algorithm first introduced in [26]. It is known as a statistical classifier as it uses the statistics of a population of data to generates a decision tree. Through supervised learning, C4.5 takes a training set of data with individuals of known classes and separates them into groups based on quantitative information taken from their traits and characteristics. C4.5 chooses each node of the decision tree based on what attribute most effectively split the data into class subsets.

Decision trees are built using a statistical property called information entropy. It uses entropy as the basis for its choices, using the difference in entropy (information gain). Information gain is a measurement of how well a feature separates training instances into target classes. The tree building process favors features which distinctly separate class members. A leaf in a decision tree indicates a class, and a decision node specifies a test to perform on a single attribute. Each outcome from the test branches to its own subtree. An example subtree of a decision created on the dataset used in this research is shown in Figure 4.7.

C4.5 has been shown to be outperformed by a genetic algorithm batch-incremental concept Learner on a 6-input multiplexer problem [27]. The problem had 6 features and each feature had 2 possible values, for a total of 64 problem instances represented by 12-bit strings. Multiplexer problems describe a k-input boolean function from input/output examples, but any single input line is not helpful in determining class membership [27]. The poor performance of C4.5 in this test was not attributed to using a decision tree for its representation, but to an information theoretic search bias [27]. This bias is created by the classification description and representation of the problem as well as the search algorithm.

4.5. C4.5 BENCHMARKING RESULTS

For a baseline comparison, C4.5 was run on the benchmarking dataset. This was implemented in the open-source data mining and machine learning suite called orange [28].

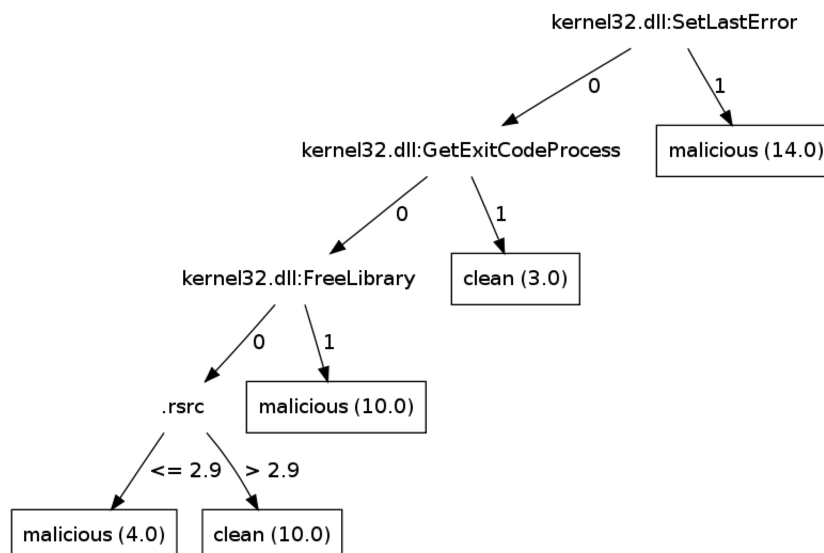


Figure 4.7: Portion of a decision tree created using C4.5

To determine how C4.5 performs on different size datasets, multiple runs of varying sizes were studied. One limitation of the C4.5 implementation in the orange framework was the size of the dataset it was able to analyze. When using large datasets, the number of features grew into the tens of thousands and C4.5 crashed while being run; this limitation could be a feature dimension memory constraint or a limit of the framework used.

One benefit of C4.5 is that it has a low computation time. After being given a problem description, C4.5 does not take long to generate a decision tree. Generating the problem description turned out to be a much longer task than the running of the algorithm. C4.5 is dependent on having a complete description of all problem instances and features. Each dataset must be organized into a table format describing all instances and attributes. Generating a table with hundreds of rows (problem instances) multiplied by tens of thousands of columns (features) is not a trivial task. Each run is averaged over a 10-fold cross-validation test, and therefore ten training and ten test tables were generated for each run. The amount of time required to generate the descriptions C4.5 uses to build trees offsets the small run-time of the algorithm itself.

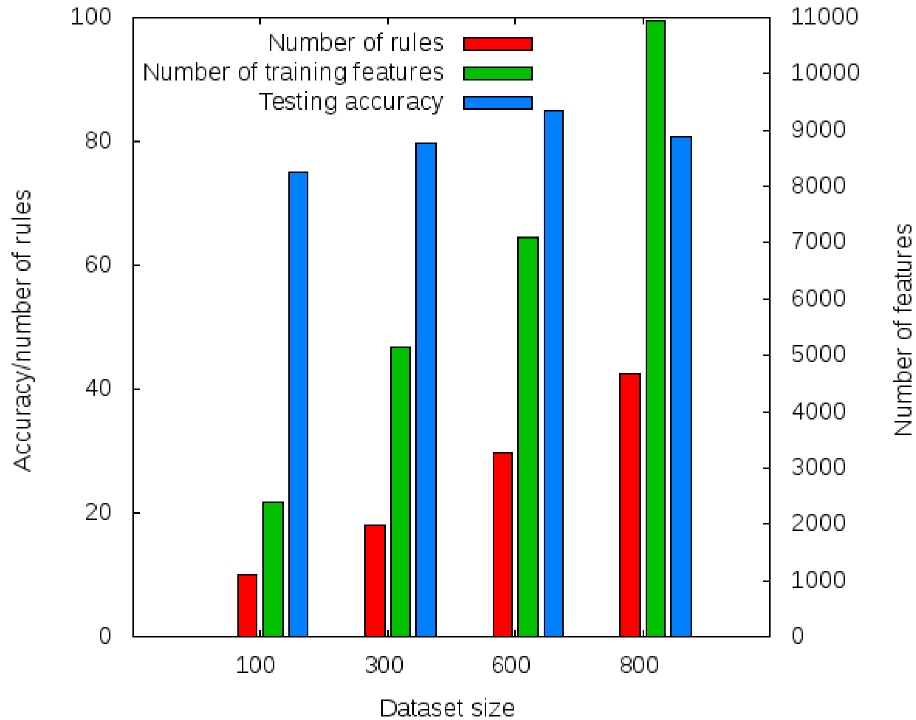


Figure 4.8: C4.5 Results

Results from using smaller datasets are presented in Figure 4.8. As the dataset size increases, the number of leaf nodes increases as well. More attribute values are required to split larger datasets. Larger datasets include a higher number of training features and they create a larger space to search through, thus the need for larger decision trees. The testing accuracy of C4.5 ranged from 75.0% on a dataset of 100 files to 85.0% on a dataset of 600 files.

C4.5 is a non-adaptive technique, and its ability to classify files does not improve over time. In order to run on different size datasets, C4.5 requires a complete description of the training dataset and attribute list. It is a batch learner and if a single new problem instance is added, the description must be re-calculated and C4.5 starts over from scratch. Adaptive techniques, on the other hand, are able to continuously improve their results as they learn a dataset. If an adaptive learning algorithm was used, specifically an incremental learner, adding a new problem instance would not require starting the algorithm over from scratch.

C4.5 was not able to parse large datasets, and table generation required a much larger amount of time than the algorithm itself took to run. C4.5 cannot reach optimal performance on the tests of the benchmark dataset. These results indicate C4.5 is not optimal for running on sparse datasets with lots of attributes. These drawbacks of C4.5 warrant an investigation of adaptive machine learning techniques that may be able to outperform C4.5.

5. EVOLUTIONARY COMPUTING

Learning classifier systems are adaptive machine learning systems from the field of evolutionary computing. They combine aspects of computer science and biology. Evolutionary computing draws inspiration from natural processes such as the Darwinist theory of natural selection. On a computer, evolutionary processes can be simulated at speeds thousands of times faster than real-time. Evolutionary computation applies Darwin's survival of the fittest principle to a population of individuals. Evolutionary computing techniques search a problem space by evolving solutions that become better over time, as they adapt to the problem.

Recent advances in technology have created a demand for automatic problem-solving as well as an increase in the complexity of problems humans are trying to solve. Algorithm design can not keep up, and there is a need for general algorithms that can be applied to a wide range of problems and still deliver acceptable, but not necessarily optimal, solutions. Evolutionary computing can fulfill this need by providing automated solutions with acceptable solutions in less time than developing a tailored algorithm design.

Evolutionary algorithms (EAs) drive the field of evolutionary computing. EAs act on a population of individuals, each of which represents a solution to the problem, and by appling environmental pressures, they simulate the biological process of natural selection and the overall fitness of the population is increased. A general evolutionary algorithm is given in Algorithm 1.

Algorithm 1 General evolutionary algorithm process

```
Initialize the population with randomly-generated candidate solutions
Evaluate the fitness of each individual
while termination condition is unmet do
   1. Competitive selection of parents based on fitness
   2. Apply genetic operators to parents to create offspring
   3. Evaluate the fitness of the offspring
   4. Remove some of the least-fit individuals from the population
end while
```

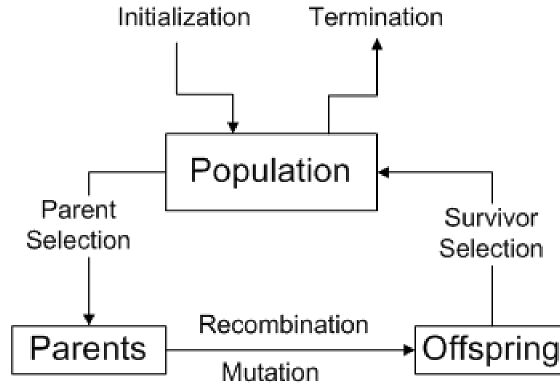


Figure 5.1: Evolution Cycle

Each generation chooses a selection of individuals to reproduce, promoting genes that represent a better solution. This determination is based on an evaluation function, called the fitness function, that grades each individual. The initial random population is graded on a fitness measure and parents are selected based on fitness. The stochastic processes of mutation and recombination to genes of parents creates children with the goal of creating individuals with a higher fitness, thus moving the population closer to an optimal solution to the problem. The new candidates compete with existing ones for a place in the next generation. A visual overview of the process is presented in Figure 5.1.

5.1. EVOLUTIONARY ALGORITHM COMPONENTS

Biological processes in nature influence individuals to become better adapted to an environment and those that survive long enough to reproduce pass on their genes, slowing evolving the entire population. There are many components of nature that contribute to these biological mechanisms, and EAs simulate many of these components. Each one contributes to how the algorithm runs.

5.1.1. Representation. The definition of an individual represents how it forms a possible solution to a problem. This maps the original problem onto a problem landscape that can be searched by the EA. The problem space is encoded into a space that can be represented by genes, which is where the evolutionary search takes place.

5.1.2. Fitness function. The fitness function (evaluation function) of an EA estimates a solution's quality. It guides the evolution process as the EA uses this measure for selection and determination of which solutions are considered better. As fitness is evaluated many times during evolution, an ideal function is closely mapped to the algorithm's goal and can be computed quickly.

5.1.3. Population. EAs process a collection of potential solutions. Individuals themselves do not evolve, the population as a whole does as better individuals are inserted and poorer ones are removed.

5.1.4. Parent selection. Parent selection chooses individuals to reproduce and pass on their genetic encodings of a solution. The selection is probabilistic to promote fitter individuals. Parent selection is responsible for improving the quality of the population.

5.1.5. Recombination. Recombination mixes information (genes) from two solutions into new ones. The parts from each parent to combine are chosen stochastically. The goal of and recombination is to combine desirable features from two individuals into a single child.

5.1.6. Mutation. Mutation in an EA slightly modifies an individual to create a mutant, much like a mutation in biology. Mutation causes a random, unbiased change in the genes of a solution.

5.1.7. Survivor selection. Survivor selection is similar to parent selection, as it chooses individuals based on their fitness, but is used after offspring have been created. The population size is generally static and survivors are selected to move on to the next generation.

5.1.8. Initialization. Initialization in an EA is generally done by random generation, but problem-specific heuristics can be used to try and create a higher fitness initial population.

5.1.9. Termination. An EA runs until a termination criterion is met. Common options are a length of time, fitness evaluations, or fitness stops improving.

5.2. GENETIC PROGRAMMING

Genetic Programming is an evolutionary computation (EC) technique that searches for computer programs solve a described problem. This automated problem solving

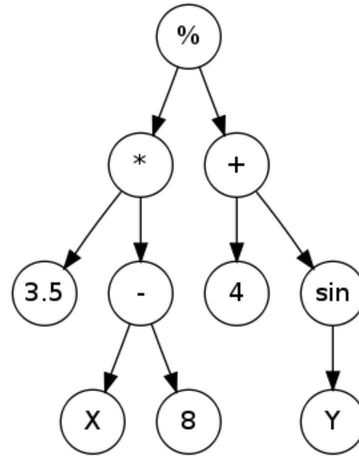


Figure 5.2: Genetic programming tree representation

method takes a high-level statement of the problem and does not require the user to specify the structure of the solution.

Genetic programming differs from other forms of evolutionary computation by using different forms of representation, initialization, genetic operators, and fitness evaluation.

5.2.1. Representation. In genetic programming, individuals are normally expressed as syntax trees, not lines of code. The variables and constants of a program are the leaves of the tree (known as terminals) and arithmetic operations are the internal nodes (functions). The set of terminals and functions make up the primitive set of a genetic program. An example representation is shown in Figure 5.2 which corresponds to the equation: $(3.5 * (\mathrm{X} - 8))/(4 + \sin(\mathrm{Y}))$.

5.2.2. Initialization. In genetic programming, two common initialization methods are *grow* and *full*. In both methods individuals are created so they do not exceed a specified maximum depth. Depth of a node is measured by counting the number of edges it takes to traverse to the node starting from the root node. The full method creates by randomly selecting nodes from the function set until the maximum depth is reached, then terminals are randomly selected. Each tree is full and all leaves are at the maximum depth. The grow method creates trees of varying size and shape. Nodes are selected from the full primitive set, unless maximum depth is reached, when only terminals can be selected.

5.2.3. Recombination. Recombination in GP differs from other evolutionary algorithms, the most common method is subtree crossover. In subtree crossover, a crossover point is randomly selected in two parents and the offspring is created by replacing the subtree rooted at the first parent's crossover point with the subtree rooted at the second parent's crossover point.

5.2.4. Mutation. The most common form of mutation in genetic programming is known as subtree mutation. This method randomly chooses a mutation point in an individual's tree and replaces the subtree rooted at that node with a randomly generated subtree. Another mutation method, known as point mutation, selects a random node and replaces the primitive with a random primitive of the same arity. This is genetic programming's equivalent of the bit-flip mutation used in genetic algorithms. Subtree mutation replaces exactly one subtree, while point mutation happens on each node with a specified probability, allowing multiple nodes to be mutated.

6. LEARNING CLASSIFIER SYSTEMS

Learning classifier systems combine reinforcement learning (RL) with evolutionary algorithms to create a rule-based expert system. The earliest learning classifier system concept was introduced by John Holland in 1976 [29] and was an expansion on his earlier invention, the genetic algorithm (GA) [30]. Holland expanded his framework into what became the standard learning classifier system. But this system was too complex to realize the intended behavior/performance [31], and Wilson created additional classifier systems which kept the original framework from Holland but simplified it to increase its performance. These new systems are discussed later in this section.

The basic framework of an LCS consists of (1) a finite population of classifiers that represents the current knowledge of the system, (2) a performance component, which regulates interaction between the environment and classifier population, (3) a reinforcement component, which distributes the reward received from the environment to the classifiers and is the learning mechanism, and (4) a discovery component which employs an EA to evolve better rules [32]. Each rule is in the form of "IF condition THEN action", and when the information from the environment matches its condition, the rule is used in the decision making process. The condition of a rule encodes when the rule is applicable to the environment and the action of a rule tells the system what action to perform.

LCSs can be applied to different problem domains including optimization problems, classification problems, and RL problems [33]. Optimization problems are solved by searching a solution space for the best solution. One issue with optimization problems is when there are local optimums, the fitness function may mislead an algorithm away from the global optimum. In a classification problem, the LCS learns to which class each problem instance belongs. Feedback is immediate and problem instances can be sampled independently. Contrary to classification problems, feedback in RL problems provides an indication of the quality of an action and may not be immediate. RL problem instances may be dependent on each other, as subsequent input depends on previous input and chosen actions [33]. Classification problems can

be redefined as single-step RL problems where reward is immediate. Multi-step RL problems have delayed reward that propagates backwards. Malware detection is a boolean classification problem, where each problem instance (a file) can be classified as one of two classes (malicious or non-malicious). This classification problem can be learned by an LCS when converted into a single-step RL problem. A classification learning system's goals are to have a high percentage of correct classifications (accuracy) while being able to classify unseen problem instances (generality) [33].

6.1. DISCOVERY

As described earlier, evolutionary algorithms or genetic algorithms drive the discovery mechanism of a LCS. New rules are created in hopes of being rewarded with a higher payoff than that of their parents. The EA in an LCS operates on a population of rules or rulesets, each of which represents a solution to the problem. The condition of each rule is encoded in its genotype and the solution is represented by the action of the rule. The evolutionary algorithm operates on individuals, mixing their genes by recombination and mutation. This process allows for new rules to enter the population and compete in natural selection.

6.2. LEARNING

In the field of artificial intelligence, learning is defined as the improvement in performance by acquiring knowledge through experience [32]. Reinforcement learning drives the evolutionary component of the LCS. In reinforcement learning, a learning system attempts to maximize reward by learning through trial and error. This approach is similar to animal learning theory, where secondary reinforces cause animals to associate external stimuli with food or pain [31].

Along with a condition and action, each rule has values associated with its fitness. The environment provides a numerical reward which guides the learning process. As the environment distributes reward, iterative updates to fitness parameters drive the LCS learning mechanism. Reinforcement learning serves two purposes: to promote individuals that obtain high rewards and to help discover better rules [32].

6.3. MICHIGAN AND PITTSBURGH STYLE LCS

After Holland first introduced the LCS, that style became known as a Michigan-style LCS [32]. The evolutionary algorithm operates on the individual level and the entire population of rules represents a solution. An alternate LCS implementation was introduced and became known as Pittsburgh-style [34]. In this type of LCS, a population consists of variable length rule sets. Each rule set is a potential solution instead of each individual. The EA operates on the level of an entire rule set in a Pittsburgh-style LCS.

Each style of LCS is best applied to a certain type of learning. Pittsburgh style LCSs are usually applied in "offline" or batch learning scenarios, where all training problems are presented simultaneously to the learner which results in a rule set that does not change over time [33, 32]. Michigan style LCSs are designed to work "online", incrementally learning each problem instance individually and evolving the rule set over time with each new observation. Offline learning is characteristic of data mining problems. This research uses a Michigan style LCS to continuously evolve a malware classifier.

6.4. OVERGENERAL CLASSIFIER PROBLEM

The concept of *strong overgenerals* in learning classifier systems is discussed in [35]. Two problems faced by learning classifier systems are overgeneral rules and greedy classifiers. A general classifier has a condition that matches many problem instances. Overgeneral classifiers advocate the desired action in some of the conditions they match, but not all of them, so performance suffers in these states. The greedy classifier problem occurs when the fitness of a classifier depends on to the magnitude of the reward it receives. Rules which match in high reward parts of the environment will reproduce often and there may not be enough rules that match low rewarding states [35]. The overgeneral and greedy rule problems combine to create problematic rules that are known as *strong overgenerals*. These rules act correctly in high reward states and incorrectly in low reward states. Strong overgeneral rules are unreliable, but outweigh reliable rules during action selection.

The problem concerns the generality versus specificity of rules. When a general rule has a higher reward prediction than a specialized rule and both match a certain

condition, a problem can arise. If the two rules have different actions, the one with higher reward will be chosen. While the general rules will be chosen since it has a higher reward prediction, the action proposed by the specialized rule may be the better choice in certain conditions.

Solutions to the problem of overgenerals include fitness sharing techniques and an accuracy-based fitness. Fitness sharing uses reward competition to eliminate strong overgenerals. Some types of LCS use a fitness based on how accurate a rule can predict its reward, and promotes those with a low variance. The strong overgeneral problem shows the importance and challenge in defining the structure of the problem to solve and the objective of the learning system.

6.5. STRENGTH-BASED AND ACCURACY-BASED FITNESS

In traditional strength-based learning classifier systems, a rule's fitness is known as its strength. The strength value is used in both action selection and reproduction. XCS introduced a new method of using different values for action selection and reproduction.

In strength-based fitness, fitness is equal to strength and is the only value used for action and parent selection. Strength is updated using reward from the environment as well as a learning rate constant.

In accuracy-based fitness, the strength parameter is still calculated using the same method but it is also used to calculate other parameters used in reproduction selection. Strength is also known as prediction, as it predicts the reward a rule will receive when it is used in the system. The other parameters used in accuracy-based fitness are based on the prediction value. Prediction error estimates the error of a rule's prediction value. The accuracy of a rule is based on an error threshold and a rule is fully accurate if its error is below that threshold. Relative accuracies are calculated for each rule in the action set. Fitness is updated to represent the average of each rule's relative accuracy.

6.6. ZCS

The *Zeroth-Level Classifier system* was originally proposed in [36]. The goal of the system was to keep the original ideas of LCS but to make a simplified version of

it, to increase its understandability as well as its performance. ZCS did not have the shortcomings of the complexity of the original LCS, yet still had good results.

ZCS removed the internal message list and rule-bidding from the original LCS framework, as well as removing unnecessary algorithmic components originally introduced to improve performance of LCS [32]. Without a way to pass information between iterations, ZCS's rules depend on how the system interacts with the environment. Each rule r in ZCS has three main components: $r = (c : a \rightarrow s)$ where:

- c is the condition which represents the environmental state the rule matches
- a is the action which the rule advocates
- s is the cumulative credit a rule has received (strength)

In every cycle of ZCS, the match set is generated to contain all individuals that match the current set of inputs of the environment. ZCS introduced the idea of using action sets, groups of rules with the same action, instead of individual rules for action selection and reinforcement. The match set is subdivided into action sets according to the action each matching rule advocates. Action selection uses a fitness proportionate selection, where fitness is the sum of the rules' strength in each action set. Figure 6.1 shows the typical structure of ZCS.

Discovery in ZCS is caused by two mechanisms, an EA and a covering operator. For each time-step, the EA has a certain probability of being invoked. It uses roulette wheel selection to choose two parents based on their fitness and creates two offspring using crossover and mutation. Rules are deleted to make room for children using roulette wheel selection based on the inverse of their fitness.

The covering operator is used to create a new rule with a condition that matches the current state of the environment when no rules in population match, or when the match-set does not contain enough high quality rules.

The original credit assignment technique in LCS was the bucket brigade algorithm (BBA). When ZCS was introduced, it introduced a new concept which merged ideas from BBA along with the Q-learning strategy used in other reinforcement learning problems [37]. Strength in ZCS moves from each action to the previous action set which rewards sequences of actions that gain reward from the environment. A discounting factor is used to create a preference for shorter chains of actions.

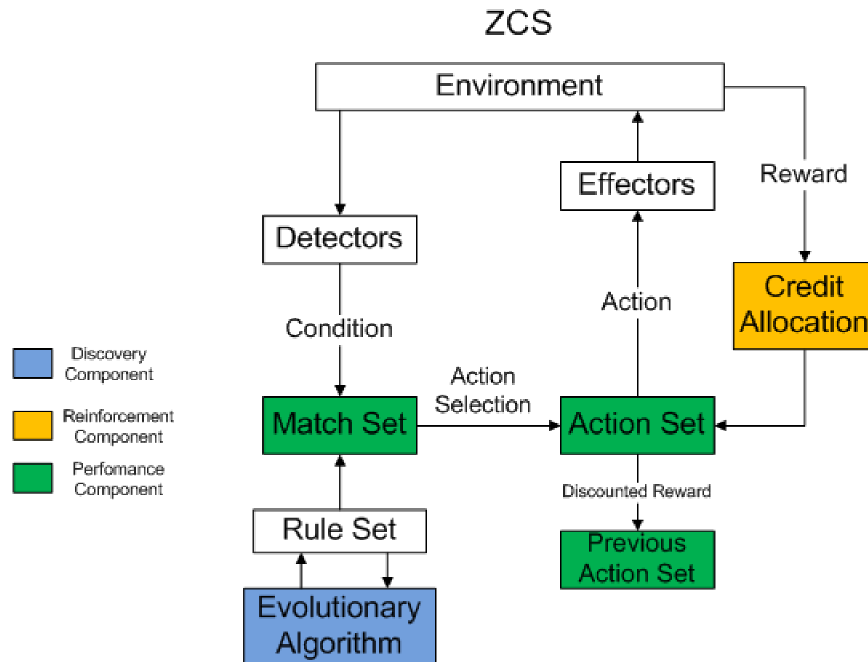


Figure 6.1: ZCS overview

The rules in the system are continually updated as new rules are created. Reward is divided among rules in each action set and the EA acts globally across the entire population, which creates fitness sharing among rules. This preserves environmental niches, action sets, but the system only preserves the rules in each niche that lead to maximum reward.

7. EXTENDED CLASSIFIER SYSTEM (XCS)

One drawback of ZCS is that it does not evolve a complete mapping of the environment from rules and their actions to received rewards, as well as recombining rules from different niches [38]. The eXtended Classifier System (XCS) was created to address these issues.

XCS addresses the problem of generalization by using a niche GA along with population-wide deletion. The specific problem of strong overgenerals is solved by changing fitness from a strength-based measurement to an accuracy-based one [33]. The new fitness is derived from the accuracy of reward prediction, instead of reward prediction itself, hence the reason XCS is known as an accuracy-based LCS. Using accuracy-based fitness enables XCS to evolve an accurate solution for all problem instances as well as a complete and accurate mapping of solutions to rewards rather than focusing only on high-payoff niches in the environment. Reinforcement learning is also known for learning a function that maps a complete representation of the state/action space, and XCS is known for successfully bridging the fields of LCS and RL [32].

Figure 7.1 shows the typical structure of the XCS algorithm. In XCS, five main components make up a rule along with additional parameters:

- Condition C represents the environmental state the rule matches
- Action A is the action which the rule advocates
- Reward prediction R estimates the average reward the rule has received
- Reward prediction error ϵ estimates the error of the reward prediction
- Fitness F represents the relative accuracy of the rule

Reward prediction R is updated iteratively to represent a moving average measure of reward received by the environment. Fitness F is a function of prediction accuracy instead of prediction magnitude and is scaled relative to other rules in each environmental niche. An illustration of a rule is shown in Figure 7.2. Separating

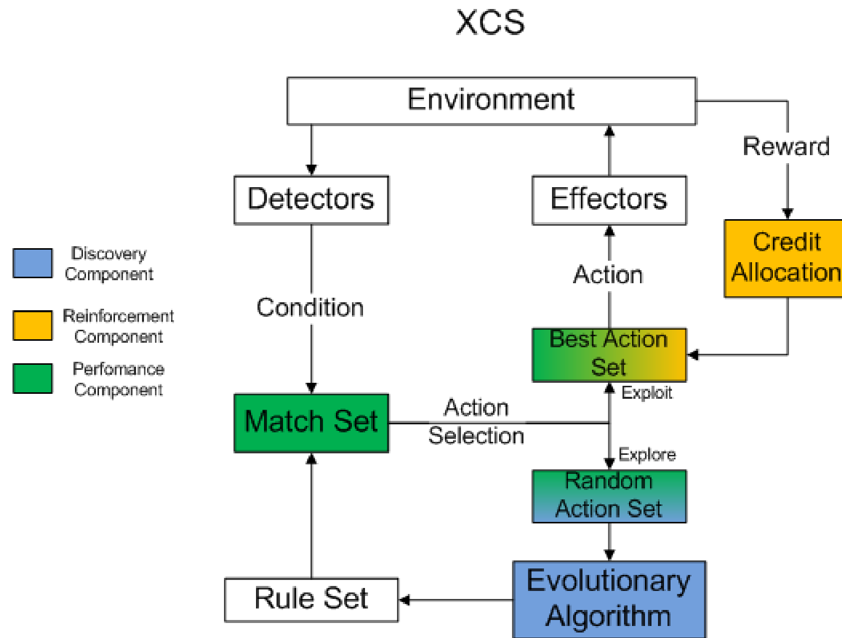


Figure 7.1: XCS overview

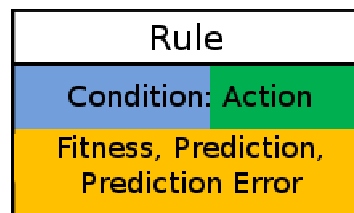


Figure 7.2: XCS rule representation

the credit assignment from the fitness used in evolution is one of the most import innovations of XCS [32].

Rules in XCS include some additional parameters not present in the original LCS:

- Action set size *as* estimates the average size of the action set a rule belongs to.
- Time stamp *ts* specifies when the rule was last included in evolution
- Experience counter *exp* keeps a record of how many parameter updates the rule has received.

- Numerosity *num* is used to combine multiple identical rules into one single rule, and numerosity is the count of rules the macro-rule represents. This speeds up computation time.

7.1. INITIALIZATION

XCS may start with either an empty population or a population of randomly generated rules, which have a random condition and action. If the population is empty, the covering operator creates a new rule for each problem instance that is not covered by the current population of rules. The difference in these two methods is studied in the results section. An advantage of initialization due to covering is that the problem space is immediately covered [33].

Population initialization needs to provide a general population in order to provide enough time for evaluation and EA application [33]. If the population size is too small or the initial population is over-specialized, this requirement is not met and XCS does not perform optimally. Covering normally occurs briefly at the start of a run until a population is established that covers all of the problem instances. If rules' conditions are too specific, XCS can get stuck in an infinite covering-random deletion cycle [33]. This cycle is due to the population being of a fixed size, and if the population is already full when covering is applied, a rule must be deleted to make room for the covering rule. If coverage is continuously triggered, the system will remove an individual, and evolutionary pressures do not have time to take affect.

7.2. EVALUATION

Rule evaluation is done every iteration and is detailed in [33]. The match set, $[M]$, contains all rules that match the current problem instance. If the match set is empty, the covering operator is applied. From the match set, a prediction array $P(A)$ is formed:

$$P(A) = \frac{\forall r \in [M] \sum\limits_{r.A=A} r.R * r.F}{\forall r \in [M] \sum\limits_{r.A=A} r.F} \tag{2}$$

The prediction array is a fitness-weighted average of all reward predictions from the rules in the match set that recommend action A. Action selection methods for XCS are usually explore, randomly selecting an action found in $[M]$, or exploit, deterministically choosing the best action within $[M]$. Typical schemes alternate between the two methods from one iteration to the next.

In classification problems, rules in action set $[A]$ are immediately updated with reward P from the environment. In single step problems P is equal to current reward, while in RL problems it equals current reward plus a discounted future reward. Parameters are updated in this order: prediction error, prediction, fitness. The update algorithm is very similar to Q-learning.

First, reward prediction error ϵ for each rule in $[A]$ is updated:

$$\epsilon = \epsilon + \beta(|P - R| - \epsilon) \tag{3}$$

Learning rate β controls accuracy and adaptivity of the moving average estimation [33]. Next, reward prediction R for each rule in $[A]$ is updated:

$$R = R + \beta(P - R) \tag{4}$$

The fitness of each rule is based on a scaled relative accuracy κ' which is calculated as follows:

$$\kappa = \begin{cases} 1 & \text{if } \epsilon < \epsilon_0 \\ \alpha(\frac{\epsilon}{\epsilon_0})^{-\nu} & \text{otherwise} \end{cases} \tag{5}$$

$$\kappa' = \frac{\kappa}{\sum\limits_{r \in [A]} r.\kappa} \tag{6}$$

κ is an estimate of a rule's accuracy that uses a power function with exponent ν to prefer low error classifiers. Threshold ϵ_0 is a maximal error tolerance, and rules with error below this threshold are considered accurate. Relative accuracy κ' is scaled with respect to the rules in $[A]$. This has the effect of fitness sharing, each rule in $[A]$ competes for a limited resource that is dependent on κ.

The fitness F of each rule is finally updated:

$$F = F + \beta(\kappa' - F) \tag{7}$$

Fitness represents the average, relative accuracy of a rule.

The action set size as is also updated in a similar way:

$$as = as + \beta(|[A]| - as) \tag{8}$$

Additionally, the reward prediction, prediction error, and action set size parameters are updated using a moyenne adaptive modifiée (adaptive average-based modification) technique [33]. This method sets the parameter value directly to the average of the values received thus far, if the experience of a classifier is less than $1/\beta$.

7.3. EVOLUTION

Genetic reproduction in XCS is performed by a genetic algorithm which runs on the current action set $[A]$. Evolution is performed if the average time since the last EA application (stored in the *timestamp* parameter of all the rules in $[A]$ exceeds threshold θ_{EA}.

Two parents are selected using proportionate selection based on rules' fitness. Two offspring are created using crossover and mutation on the parents. Offspring parameters are initialized by averaging the parent's fitness F, prediction R, and prediction error ϵ. The fitness F of offspring is decreased to 10%, being pessimistic about the offspring [33]. Offspring are not assumed to have high fitness based on the performance of their parents. Experience exp is set to 1.

If the population size is greater than the specified maximum, extra rules are deleted with a probability proportional to their action set size parameter as. If a rule has experience greater than threshold θ_{del} and a fitness lower than a fraction (parameter δ) of the average fitness $\bar{F}$ of rules in the population, its probability of being deleted is multiplied by $\bar{F}/F$, increasing the chance of removal of the lower fit individual.

8. SYSTEM DESIGN

The learning classifier system this thesis is based on is a variant of the XCS model. It has been adapted for use in the problem domain of malware detection, including replacing the bit string representation with s-expressions. An overview of this learning classifier system is shown in Figure 8.1.

The training and test sets come from the pre-processing component of the system. Once all the executables are analyzed and features are extracted, the learning classifier system takes the feature list and evolves rules until a termination condition is met.

8.1. LEARNING CLASSIFIER SYSTEM

8.1.1. Rule Representation. Traditional LCS and XCS implementations operate on bit-string representations of problems. These fixed-length bit-string representations are not optimal for searching problem spaces which are described symbolically or for problems with both syntactic and semantic constraints, which can vary in length and complexity [27].

In this system the condition part of each rule is represented as an s-expression. S-expressions can be visualized as a tree structure where internal nodes are one of the logic operators $\{AND, OR, NOT\}$ and leaf nodes (terminals) are a single feature (an example is shown in Figure 8.2). S-expressions allow for more complex forms of logical relationships to be expressed over bit-strings; it is an assumption of this research that malicious files contain complex relationships of features that are identifiable from non-malicious files.

Since malware detection is a binary classification problem, the action of each rule represents whether the file is classified as malicious or not.

8.1.2. Initialization. There are three methods of rule initialization in this system: random, covering, and from a C4.5 decision tree.

During random initialization, each tree is built from the top down. Every node has a chance of becoming a function or terminal node, the *operator rate* parameter is the probability of a node becoming a function node (operator). If it is, an operator

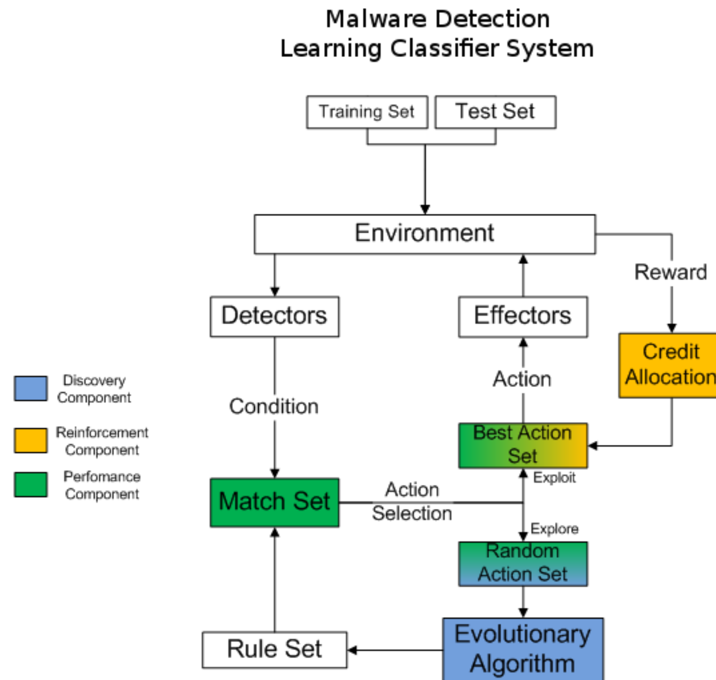


Figure 8.1: Malware LCS Diagram

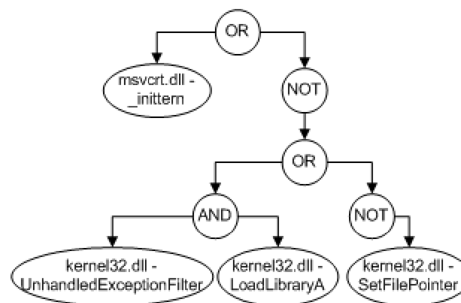


Figure 8.2: Visualization of a subtree of a generated rule

is randomly chosen from the list of operators, and the appropriate number of child nodes are created in the same fashion. If a tree is at the maximum tree height, the node becomes a terminal node.

Initialization due to covering creates a rule when the current population does not match a presented file. This enables the system to develop a complete coverage of the training set, and recombination will allow the rules to generalize the dataset. While the covering operator is used exclusively in this technique, it also available in

the other initialization methods; it is invoked if a problem instance is not matched by any rules.

A third initialization technique is generation from a C4.5 decision tree. The tree can be converted to a ruleset, similar to the method used in [15]. Each leaf node in the decision tree becomes an individual rule in the population. The initial set of rules fully represents the decision tree, but is in a format that can be evolved by the learning classifier system. By starting with the knowledge of the C4.5 algorithm, the initial population begins with a higher fitness and accuracy.

8.1.3. Action Selection. A single problem instance in this system is a randomly chosen file that is presented to the system. Each rule in the population is a parse tree, and is compared to the extracted features from the file. If the parse tree matches the feature, it is put into the LCS's match set. If no rules match a given malicious file during training, a covering operator creates a rule that has a matching condition and inserts it into the population with a chance of spreading its genetic material to offspring, allowing the population to classify the file. From the match set, an action is chosen.

There are various methods of choosing an action, and XCS typically alternates between two methods in an explore/exploit scheme [32]. Explore randomly selects an action from within the action set, while exploit deterministically selects the action with the highest reward. Once an action is chosen, the action set becomes the rules in the match set that advocate the chosen action. The evolutionary algorithm is only run during exploration and the fitness of rules is only evaluated during exploitation.

As seen in the system diagram in Figure 8.1, exploration combines the performance component with the discovery component. New rules are created by the EA and the system explores the search space in hopes of finding better rules.

Exploitation combines the performance component with the reinforcement component. The action with the highest reward is chosen and the rules are rewarded if the action was correct and punished if it was incorrect. Exploitation reinforces rules that receive reward and evaluates the rules discovered during exploration.

This combination of methods allows the system to explore a large search space while rewarding rules that lead to environmental reward. By alternating between the two methods, the system spends equal amounts of time discovering and reinforcing, thus creating a balance between the two.

8.1.4. Rule Evaluation. The rules in the action set are updated every iteration with reward from the environment. In the classification problem of malware detection, environment feedback is immediate and based on whether the action selection was correct. In XCS, the RL technique is based on the Q-learning algorithm [39]. Three parameters are adjusted to determine the performance of a rule, in the order: prediction error, prediction, fitness [33].

A feature is considered malicious or benign based on whether the file it was extracted from was identified as malicious or benign. It is important to realize that not all features of a malicious file are themselves malicious; many malicious files have benign actions and there is an overlap between the sets of benign features and malicious features. Reward is based on whether the chosen action matches the classification of the file.

8.1.5. Mutation. Mutation introduces random variation to individuals by selecting a random mutation point. The subtree rooted at that node is replaced with a randomly generated tree. The height of the generated subtree is limited so the entire tree does not exceed the *maxheight* parameter, in order to keep processing time from growing indefinitely. The terminal nodes were chosen using proportionate selection from the vector of features extracted from the training set. Features were chosen using roulette-wheel selection, with a probability proportional to the number of times a feature appeared in the training set.

8.1.6. Crossover. Crossover works similar to that in Genetic Programming. A subtree starting at a random node is chosen from each parent, and the subtree from one parent replaces the subtree from the other parent.

While crossover of individuals represented by bit strings typically produces two offspring, subtree crossover in genetic programming normally produces a single child. While subtree crossover can produce two offspring, it is not commonly used [40]. The effects of the number of children produced is studied in the results section.

During recombination the second parent's random node selection was limited to those nodes which would keep the heights of the offspring below the *maxheight* parameter.

8.1.7. Evolutionary Algorithm. In XCS, the EA reproduces rules in the action set, realizing implicit niching [41] as opposed to panmictic reproduction, where

rules are selected from the entire population. XCS performs genetic reproduction if the average time since the last EA invocation of the rules in the action set exceeds threshold θ_{EA}.

The mechanism used for parent and survivor selection is tournament selection. Parent selection chooses a set of classifiers at random, and the one with the highest fitness is chosen to become a parent. In XCS an effective method for determining parent tournament size is to make it proportional to action set size [33]. Parameter τ represents the proportion of the action set that is used in the tournament. Since the EA acts on individuals in the action set, selective pressure is based on the size of the action set. If selection pressure is too weak, learning may not take place and if selection pressure is too strong, crossover never has any effect since identical individuals are crossed [33]. Survivor selection repeatedly executes a tournament of a user specified size and deletes the least fit until the population size has been reduced to its specified size. An age requirement has to be met before rules are considered for parent and survival selection; a rule is not eligible for selection until the system has presented it with θ_{age} files.

8.2. DECISION TREE INITIALIZATION

8.2.1. C4.5 rule initialization. One of the additions to traditional XCS that this system includes is an initialization operator that takes a decision tree and generates a corresponding ruleset. This enables the system to start with knowledge about the problem and quickly narrow in on some of the optima in the problem space. These may be local optima that the decision tree has found, and as with any EA, the LCS faces challenges in finding the global optimum.

Each leaf node in the decision tree becomes a single rule in the LCS. There is a one-to-one correspondence of leaf nodes and rules. This initialization method combines the decision tree's quick run-time with the benefits of an adaptive, evolving ruleset. This approach quickly guides the LCS in the direction of high performing rules, with hopes of evolving an even better population of rules. A potential downside of this method is that the rules generated from C4.5 may guide the evolution process towards a sub-optimal solution because C4.5 may overfit to the training instances.

8.2.2. Specialization function. The number of rules generated from a decision tree is equal to the number of nodes in that tree. The population size of the LCS is static, and there may be space in the population for more rules than the decision tree contained. An additional variation function was created for decision tree initialization. The set of rules the decision tree created is specialized by adding additional features to the tree condition of each rule. This allows a single rule from the decision tree to be modified multiple times and inserted into the LCS population.

A subtree is generated and appended to the original decision tree rule. The height of this subtree is controlled by the parameter $SpecializationHeight$. This parameter controls how specialized the rule becomes, and the smaller it is, the closer the specialized rule is to the original node in the decision tree.

8.3. PERFORMANCE METRICS

The goal of this system is to evolve rules using RL that will identify malware. The pre-processing step first extracts features from the files and this collection of features is divided into training and testing sets. The LCS evolved rules over the training set, then evaluated them over the testing set.

Experiments were run using a stratified ten-fold cross-validation test. Each fold contained 50% malware and 50% goodware.

A rule's classification accuracy on a file maps to one of the following four categories:

1. True positive (TP): correctly classifies a malicious executable as malicious.

2. False negative (FN): incorrectly classifies a malicious executable as non-malicious.

3. True negative (TN): correctly classifies a non-malicious executable as non-malicious.

4. False positive (FP): incorrectly classifies a non-malicious executable as malicious.

Three different metrics were tracked: (1) Classification Accuracy, (2) Detection Rate (DR), and (3) False Alarm Rate (FAR). These are defined mathematically as:

$$\text{classification accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

$$\text{detection rate} = \frac{TP}{TP + FN} \tag{10}$$

$$\text{false alarm rate} = \frac{FP}{FP + TN} \tag{11}$$

The classification accuracy rates how the system performs in general, while detection rate and false alarm rate show more specific metrics on the trade-offs between malware coverage and false positives.

8.4. DEFAULT PARAMETERS

A list of default parameters the LCS uses in the experiments described in Section 9, unless otherwise specified, is presented in Table 8.1. There are parameters that relate to the EA, the LCS, and the individual rules.

A description of each parameter is provided next. More details on how the XCS parameters are used are described in Section 7; the core set of these parameters are the same ones as used in [33] with additional system-specific parameters.

- Population Size (μ) - Number of rules in the population
- Crossover Rate (χ) - Chance of using crossover to produce offspring
- Mutation Rate - Chance of using mutation to produce offspring
- Offspring Size (λ) - Number of offspring created per invocation of the EA
- Survivor Tournament Size - Number of individuals used in the tournament for deleting excess rules
- Accuracy Parameter (α) - Accuracy power function coefficient; differentiates accurate from inaccurate rules
- Accuracy Parameter (ν) - Accuracy power function exponent; controls the rate of decrease of a rule's accuracy as a function of its prediction error ϵ

Table 8.1: Classifier system parameters

Parameter Name	**Parameter Value**
EA parameters	
Population Size (μ)	400
Crossover Rate (χ)	1.0
Mutation Rate	0.04
Offspring Size (λ)	1
Survivor Tournament Size	5
XCS parameters	
Accuracy Parameter (α)	0.1
Accuracy Parameter (ν)	5.0
Error Threshold (ϵ_0)	10
Learning Rate (β)	0.2
Parent Tournament Proportion (τ)	0.5
EA Threshold (θ_{EA})	45
Fitness Parameter (δ)	0.1
Deletion Threshold (θ_{del})	20
Rule parameters	
Initial Tree Height	5
Max Tree Height	8
Operators	['and', 'or', 'not']
Operator Rate	0.75
Import Rate	.9
Specialization Height	1

- Error Threshold (ϵ_0) - Threshold of maximal error tolerance; provides a tolerance of noise
- Learning Rate (β) - Controls the balance between new information and old information; a large β results in a large change in the parameter estimate, while a small β results in a small change
- Parent Tournament Proportion (τ) - Proportion of the current action set that is used for tournament selection by the EA; this ensures relatively strong selection pressure, adapting to the current action set size
- EA Threshold (θ_{EA}) - Frequency of the EA application

- Fitness Parameter (δ) - Fraction of mean fitness of the population; if a rule has a lower fitness, its deletion probability is increased
- Deletion Threshold (θ_{del}) - Minimum experience threshold for fitness influence in deletion
- Initial Tree Height - Maximum size of a rule's condition during initialization
- Max Tree Height - Maximum of a rule's condition during mutation and crossover
- Operators - The list of operators available for internal nodes of a rule's condition
- Operator Rate - The probability of choosing an operator at any given node in a tree
- Import Rate - The probability of choosing an import (versus a section) for a given leaf node
- Specialization Height - The height of a subtree to add to rules created by C4.5 Initialization

9. EXPERIMENTAL RESULTS

Various datasets were analyzed and compared using different techniques, initializations, and parameters. All experiments were run using a stratified ten-fold cross-validation test. Each fold contained the same amount of malicious files (50%) and non-malicious files (50%).

9.1. FAMILY OF MALWARE STUDY

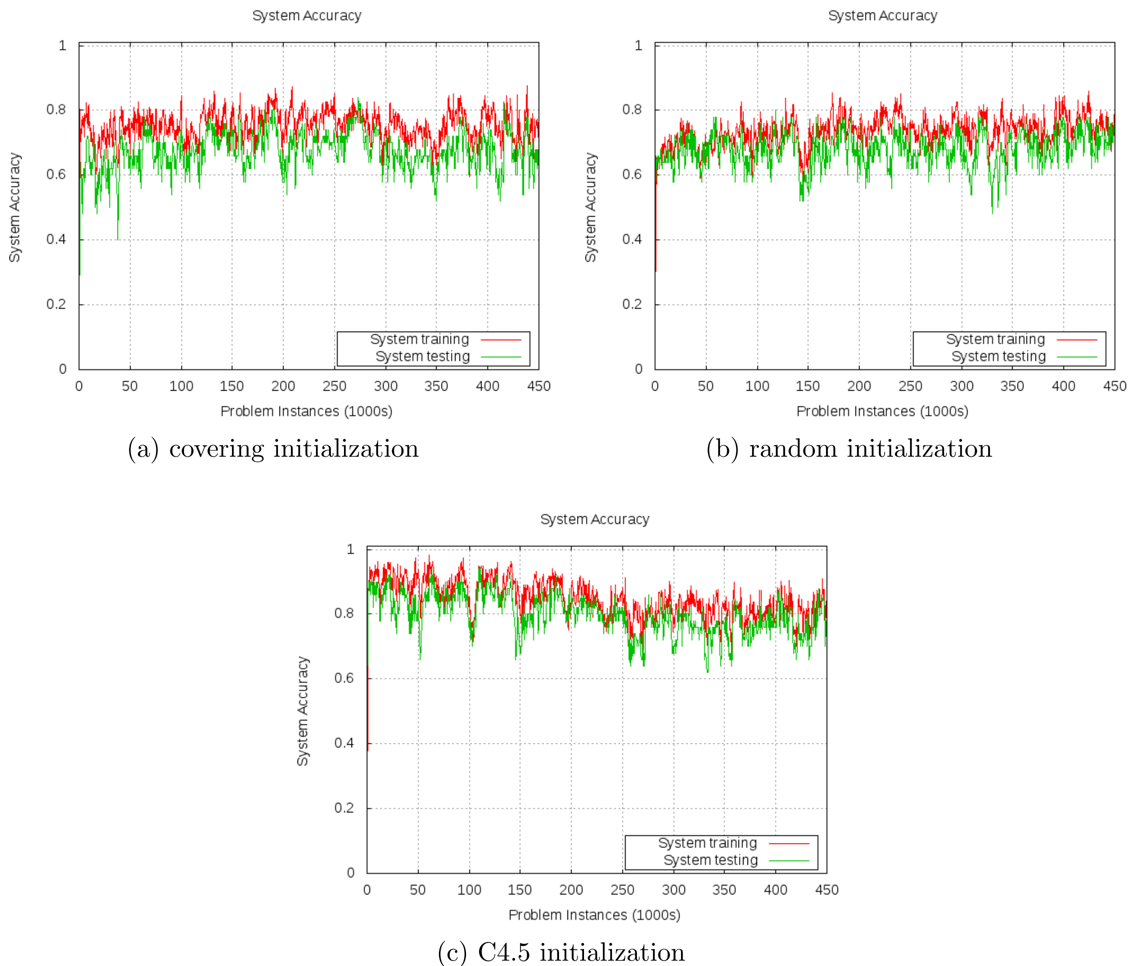
In order to determine the effectiveness of the custom LCS system, a small dataset was used in this experiment. The dataset consisted of malicious files all belonging to a family of malware known as Poison Ivy. Three initialization methods were used in this experiment: random initialization, covering initialization, and initialization from C4.5 decision trees. The C4.5 algorithm itself was used as a baseline for performance comparison. The four different results are compared and discussed for various population sizes.

For random initialization, each rule was randomly generated using the grow algorithm. For covering initialization, the population started empty, and rules were created for each file that was presented and not covered by the current set of rules. With C4.5 initialization, rules were taken from a generated C4.5 decision tree and then modified using the specialization function.

Population size was adjusted to determine how it affects accuracy as well as the minimal number of rules required to accurately classify the dataset. The dataset consisted of 50 files, and was divided into 10 folds for stratified cross validation, each fold has the same number of non-malicious files and malicious files so that each test set has identical distributions of malware and goodware.

The results are presented in groups based on what parameter was changed between runs. For each population size, the three initialization methods are compared.

Figure 9.1 shows results from using a population size of 10. A population size of 10 was too small for this dataset as evident in the figures, no convergence was achieved. The system had entered a covering-random deletion loop as described earlier. The population size was too small to adequately cover the entire set of files, so as the



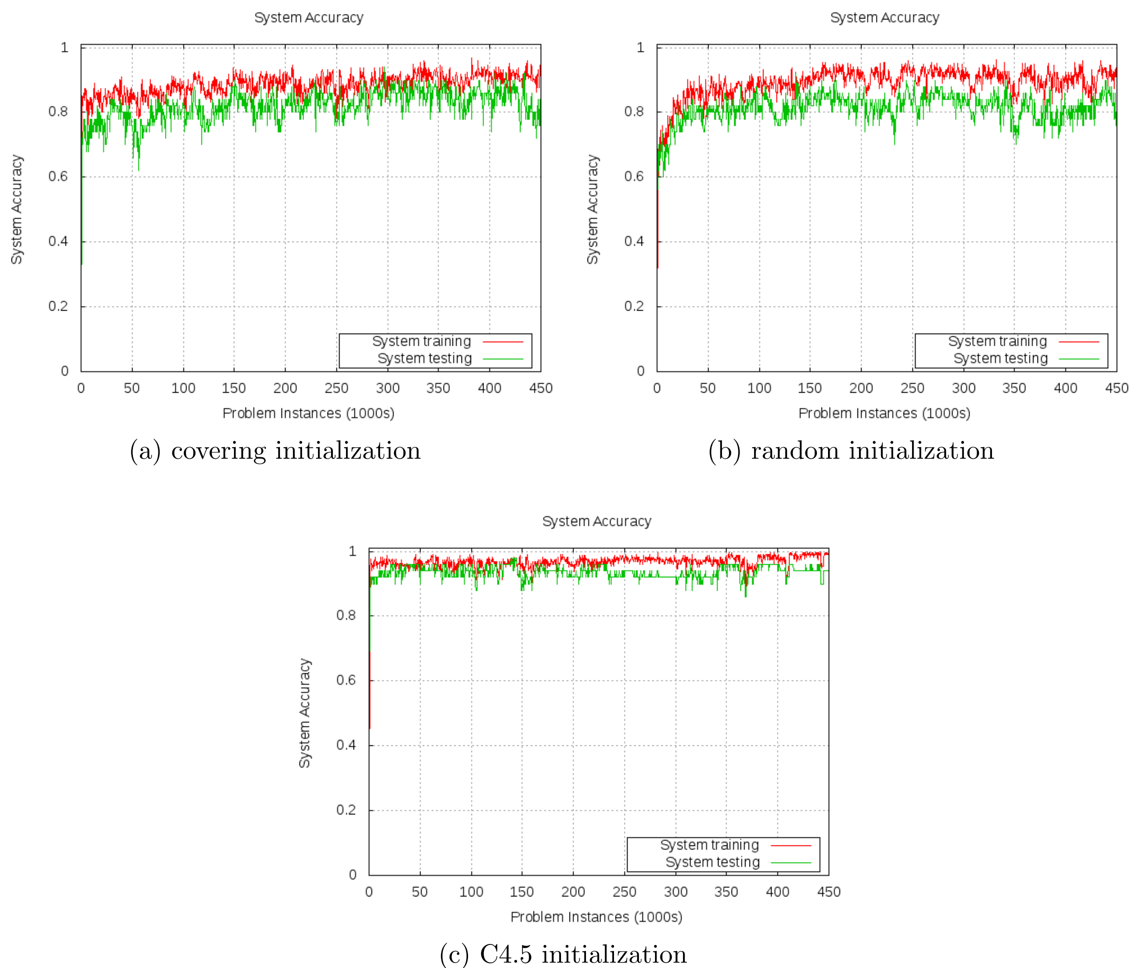
(a) covering initialization

(b) random initialization

(c) C4.5 initialization

Figure 9.1: Malware family results using population size=10

algorithm created rules to cover problem instances, it had to delete a rule already in the population, and the deleted rule created a new gap that had to be covered on the next round of file presentations. The large variance in both training and testing accuracies show the system thrashing through new individuals, unable to keep high performers in the population. The system can not cover the entire dataset, only a window of it, and as that window continuously shifts around, the performance does as well. The C4.5 initialization test started with a population of high fitness, but as it made room for offspring, it deleted some high performing rules and performance suffered as the system dropped into the same random cycle as the other methods.



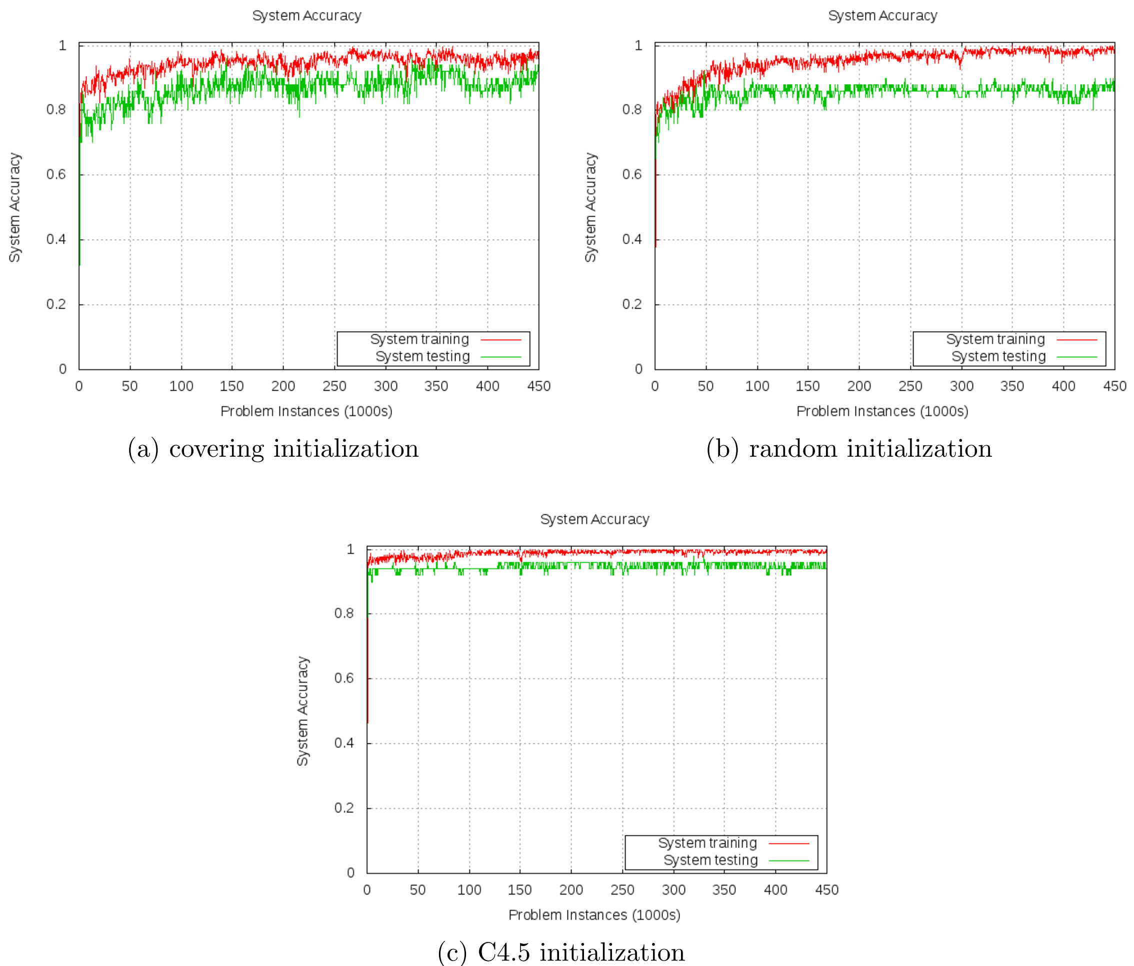
(a) covering initialization (b) random initialization

(c) C4.5 initialization

Figure 9.2: Malware family results using population size=20

Figure 9.2 shows results from using a population size of 20. This size population does better than the previous experiment, as there is a lot less variance and an increasing trend can be seen in the graphs. The performance is not quite optimal, as there still is some variation in the performance. Although the problem space was adequately covered, there were not enough individuals in the population to smooth out the performance of the system on the harder to classify instances.

Figure 9.3 shows results from using a population size of 30. This experiment had the best results, and could accurately represent the training set. The overall best accuracy the system achieved was 99.6% training and 94.0% testing using rule generation from the C4.5 decision tree with a population of 30. Closely following



(a) covering initialization

(b) random initialization

(c) C4.5 initialization

Figure 9.3: Malware family results using population size=30

was random initialization with 98.6% training and 92.0% testing. This experiment had a relatively small search space and the LCS was able to evolve rules that were competitive with those starting from C4.5.

Table 9.1 shows the averages and standard deviations from all runs in this experiment. Two-sample F-tests for equal variances and corresponding two-tailed t-tests using $\alpha = 0.05$ were employed to compare results. Not all results were statistically significant. The p values for some tests were small enough to reject the null hypothesis that the data between two runs are independent random samples from normal distributions with equal means and equal but unknown variances. Significant conclusions from the testing results can be made as follows:

- All methods improved by increasing the population size from 10 to 20.
- *C4.5* initialization improved by increasing the population from 10 to 20.
- *C4.5* initialization performed better than *random* initialization at population size of 20.
- *C4.5* initialization performed better than *covering* initialization for population sizes of 10 and 20.
- At a population size of 10, no LCS method had a significant advantage over any other.
- The C4.5 decision tree performed better than all other methods at a population size of 10.
- The C4.5 decision tree performed better than *covering* and *random* initialization at a population sizes of 20.
- The C4.5 decision tree did not perform significantly better than any LCS method at a population size of 30.
- The overall best training method was *C4.5* initialization with population size of 20.

The C4.5 algorithm was ran on the dataset and achieved 98.0% training accuracy and 96.0% testing accuracy, better than the LCS at small population sizes. At a large enough population, the LCS performed comparably to C4.5. The best performing initialization method was initializing with rules generated from the C4.5 decision tree; showing that seeding the algorithm with prior knowledge about the data allowed it to perform better than starting at random locations in the search space.

Better tuning of the LCS may enhance its performance as well as evolving it for longer periods of time. While the decision tree was able to accurately classify this small dataset, as shown in the benchmark it does not perform as well on more

Table 9.1: Family of malware study experimental results

Dataset	Initialization	μ	System Accuracy	DR	FAR
Training	Covering	10	0.727(0.141)	0.685(0.356)	0.242(0.263)
		20	0.909(0.077)	0.814(0.197)	0.032(0.075)
		30	0.964(0.064)	0.937(0.146)	0.017(0.053)
	Random	10	0.805(0.127)	0.595(0.375)	0.056(0.097)
		20	0.918(0.085)	0.899(0.138)	0.068(0.090)
		30	0.986(0.031)	0.980(0.043)	0.009(0.029)
	C4.5	10	0.855(0.123)	0.878(0.270)	0.174(0.169)
		20	**0.996(0.014)**	0.991(0.029)	**0.000(0.000)**
		30	0.986(0.031)	**1.000(0.000)**	0.022(0.051)
	Decision Tree	n/a	0.980(0.007)	0.955(0.024)	0.004(0.012)
Testing	Covering	10	0.640(0.126)	0.450(0.438)	0.233(0.316)
		20	0.760(0.207)	0.600(0.516)	0.133(0.172)
		30	0.920(0.140)	0.800(0.350)	0.000(0.000)
	Random	10	0.780(0.175)	0.500(0.471)	0.033(0.105)
		20	0.800(0.211)	0.700(0.422)	0.133(0.172)
		30	0.920(0.140)	0.850(0.337)	0.033(0.105)
	C4.5	10	0.800(0.163)	0.800(0.422)	0.200(0.233)
		20	0.940(0.097)	**0.900(0.211)**	0.033(0.105)
		30	0.940(0.097)	**0.900(0.211)**	0.033(0.105)
	Decision Tree	n/a	**0.960(0.084)**	**0.900(0.211)**	**0.000(0.000)**

diverse datasets. The next section analyzes how the two algorithms perform on a larger dataset consisting of many types of malware.

9.2. BENCHMARK DATASET COMPARISON

The family of malware dataset is an unrealistic representation of an actual malware detection system, which has to identify threats of all kinds, not just a single type of malware. This experiment tested the algorithms on the larger, more diverse dataset used for the benchmarking baseline set up in Section 4. The dataset size was 300 files and used a stratified ten-fold cross-validation test.

9.2.1. Initialization method study. For the first part of the experiment, similar tests to those run in the family of malware study were run to analyze how the algorithms performed on a larger and more difficult dataset. This includes adjusting the population size as well as the initialization method. The resulting means and standard deviations are shown in Table 9.2.

Table 9.2: Initialization study experimental results

Dataset	Initialization	μ	System Accuracy	DR	FAR
Training	Covering	50	0.837(0.048)	0.887(0.073)	0.214(0.047)
		100	0.830(0.040)	0.866(0.093)	0.210(0.103)
		200	0.907(0.022)	0.926(0.029)	0.112(0.031)
		300	0.905(0.030)	0.940(0.023)	0.127(0.056)
	Random	50	0.796(0.036)	0.865(0.093)	0.274(0.063)
		100	0.841(0.040)	0.879(0.055)	0.197(0.060)
		200	0.852(0.029)	0.921(0.044)	0.216(0.055)
		300	0.889(0.028)	0.924(0.049)	0.148(0.037)
	C4.5	50	0.897(0.020)	0.945(0.031)	0.151(0.049)
		100	0.927(0.015)	0.971(0.031)	0.117(0.043)
		200	0.962(0.013)	0.978(0.016)	**0.056(0.026)**
		300	**0.963(0.020)**	**0.987(0.015)**	0.061(0.033)
	Decision Tree	n/a	0.948(0.008)	0.973(0.009)	0.077(0.015)
Testing	Covering	50	0.790(0.092)	0.840(0.155)	0.260(0.119)
		100	0.803(0.053)	0.873(0.106)	0.267(0.144)
		200	0.813(0.063)	0.833(0.072)	**0.207(0.091)**
		300	0.783(0.071)	0.807(0.139)	0.240(0.078)
	Random	50	0.743(0.108)	0.820(0.281)	0.333(0.220)
		100	0.780(0.101)	0.807(0.219)	0.247(0.141)
		200	0.813(0.061)	0.873(0.097)	0.247(0.118)
		300	**0.833(0.074)**	0.873(0.119)	0.207(0.135)
	C4.5	50	0.800(0.061)	0.840(0.134)	0.240(0.110)
		100	0.803(0.051)	**0.887(0.077)**	0.280(0.125)
		200	0.797(0.060)	0.847(0.063)	0.253(0.133)
		300	0.790(0.072)	0.847(0.083)	0.267(0.141)
	Decision Tree	n/a	0.797(0.058)	0.847(0.083)	0.253(0.147)

Two-sample F-tests for equal variances and corresponding two-tailed t-tests using $\alpha = 0.05$ were employed to compare results. Not all results were statistically significant. The p values for some tests were small enough to reject the null hypothesis. These following significant conclusions can be made from the training results:

- *Covering initialization* performed better than *random initialization* at population sizes of 50 and 200.
- *Covering initialization* performed better at a population size of 200 over 100.
- *Random initialization* performed best on a population size of 300

- *C4.5 initialization* performed better than *random initialization* on all population sizes.
- *C4.5 initialization* performed better than *covering initialization* on all population sizes except at 300.
- *C4.5 initialization* performed increasingly better on each population size increase except for 300.
- The C4.5 decision tree did not perform significantly better than any tuned LCS method.

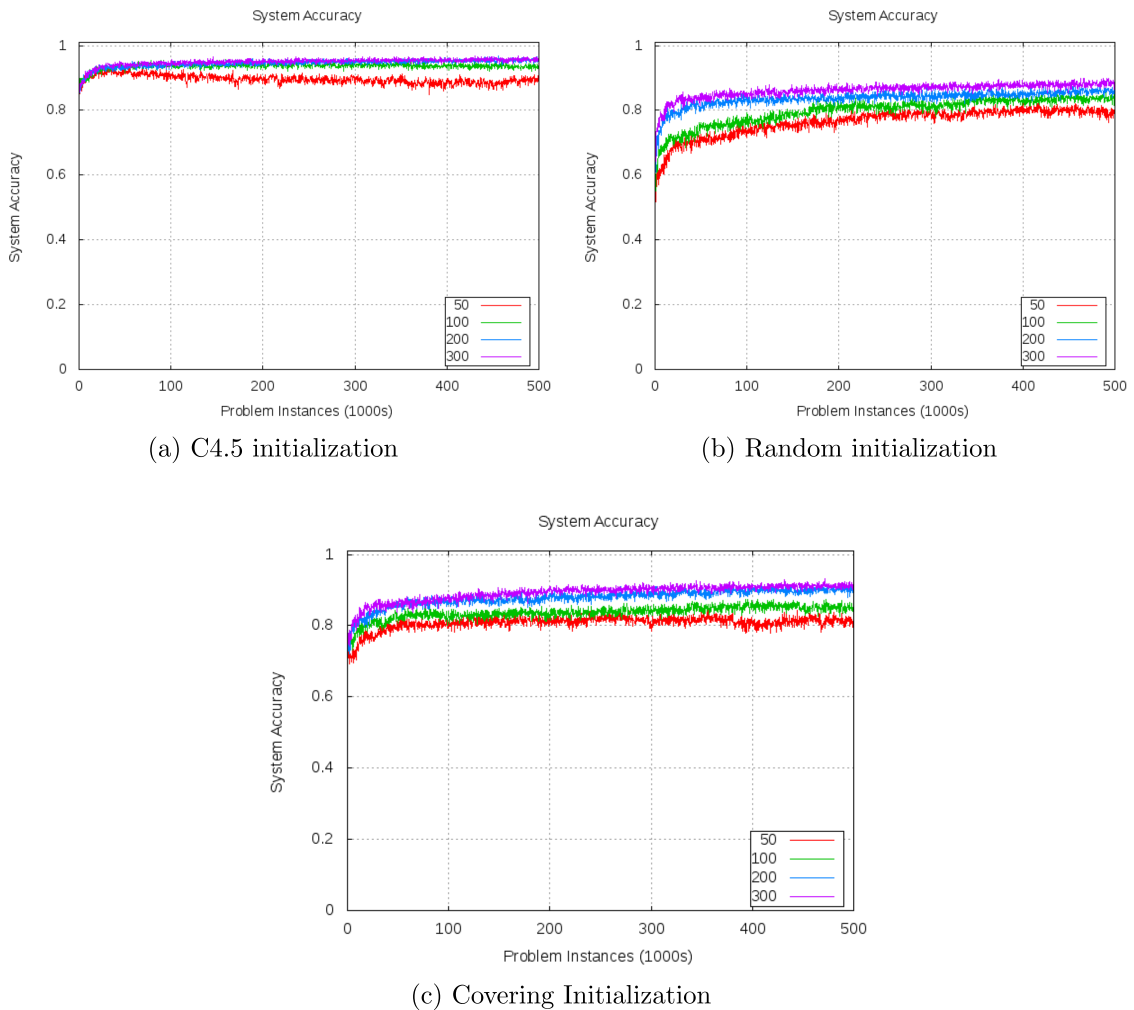
Initialization from C4.5 obtained the best system performance and this is most likely due to the fact that the system started with prior knowledge and had a coverage of the problem space from the beginning.

9.2.2. Population size study. The data in Table 9.2 is grouped by initialization method, but it can also be analyzed by population size. In the second part of this experiment, a graph was generated for each initialization method and plotted according to population size. The training accuracies of all three methods using 4 different population sizes can be seen in Figure 9.4, and testing results in Figure 9.5.

As the population size increased, there was a general trend of increased training accuracy in each method. Testing accuracies did not exhibit the same trend. While the random initialization method has a slight increase in accuracy, it is not statistically significant. A larger population allowed the classifier system to create more rules, which enabled the system to accurately classify more of the training set but not of the testing set.

Increasing the population size will not necessarily make the system test performance better, as well as having the disadvantages of making the system harder for a human to interpret and slowing down the speed of the system, as more calculations are required.

9.2.3. Offspring size study. The third part of the experiment analyzed how adjusting the number of offspring produced affected the performance of the system. The same benchmarking dataset is used as before.



(a) C4.5 initialization

(b) Random initialization

(c) Covering Initialization

Figure 9.4: Population size training results

In genetic programming one or two offspring can be created from the crossover operation. The more common method is to produce a single offspring per evolution cycle [40]. This test compared the differences between creating one and two offspring per invocation of the evolutionary algorithm. As the reader will recall, the EA is invoked based on the average time since it was last ran, so there is no set number of children or EA applications for a given number of problem instances. Averages and standard deviations are shown in Table 9.3. Changing offspring size did not have a major affect on performance. Whether one individual or two were created, evolutionary pressures still applied in the same manner.

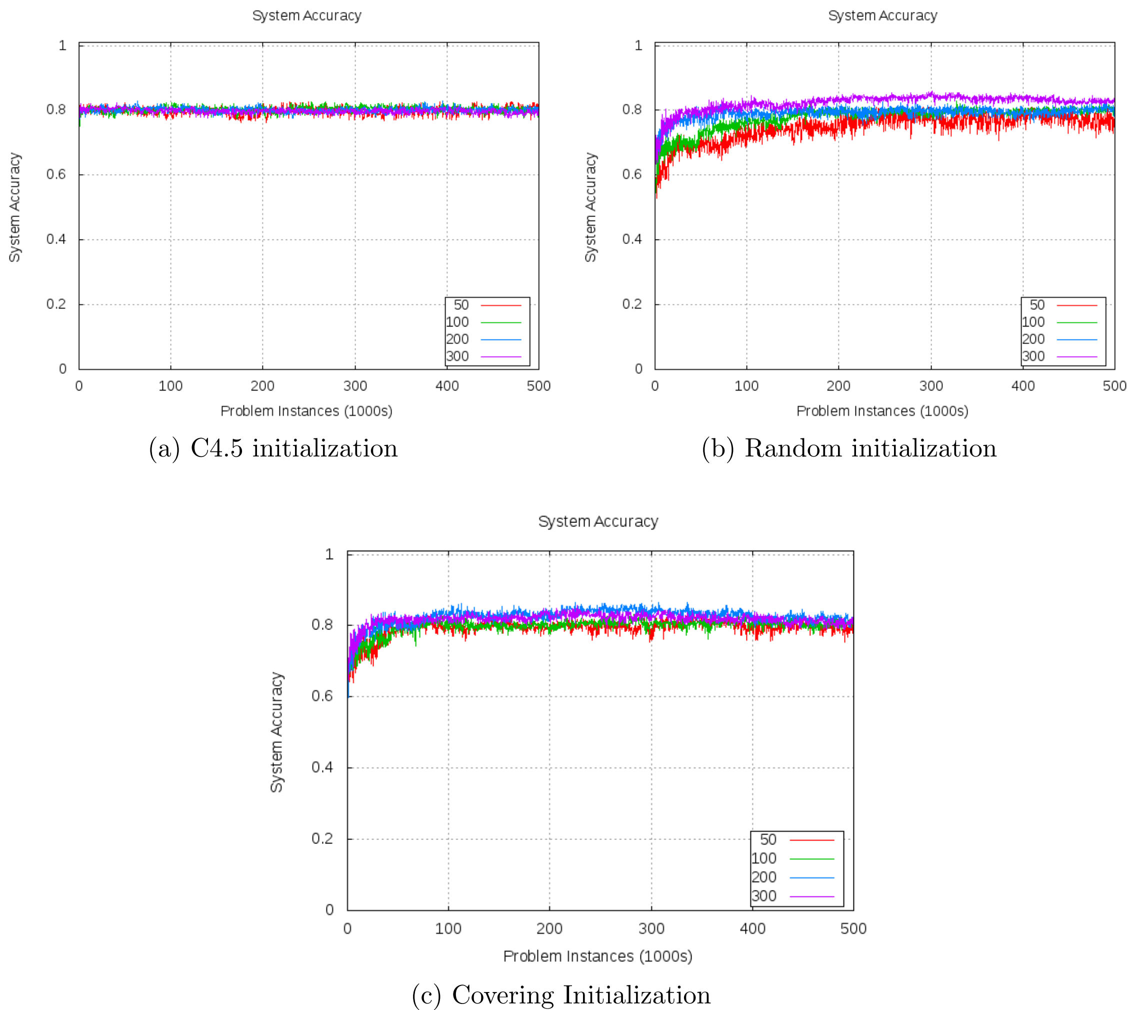


Figure 9.5: Population size testing results

9.3. SYSTEM COMPARISON TO C4.5 DECISION TREE

The datasets in these experiments came from the Offensive Computing collection, which contains user-submitted malware. The malware in this set is of unknown origin and could belong to any family, unlike the previous experiment. There is no known relationship between any two malicious files. This is an accurate representation of the real world, where internet users are potentially exposed to every file on the internet and may encounter any number of threats.

C4.5 was ran on the dataset using the orange machine learning framework [28]. The high-level structure of a decision tree is shown in Figure 9.6 and it is highly

Table 9.3: Offspring size study experimental results

Dataset	λ	μ	**System Accuracy**	**DR**	**FAR**
Training	1	50	0.837(0.048)	0.887(0.073)	0.214(0.047)
		100	0.830(0.040)	0.866(0.093)	0.210(0.103)
		200	0.907(0.022)	0.926(0.029)	**0.112(0.031**
		300	0.909(0.030)	0.939(0.048)	0.119(0.057)
	2	50	0.821(0.035)	0.891(0.054)	0.248(0.046)
		100	0.848(0.020)	0.901(0.066)	0.204(0.057)
		200	0.888(0.032)	0.949(0.029)	0.169(0.072)
		300	**0.918(0.023)**	**0.962(0.036)**	0.126(0.048)
Testing	1	50	0.790(0.092)	0.840(0.155)	0.260(0.119)
		100	0.803(0.053)	0.873(0.106)	0.267(0.144)
		200	0.813(0.063)	0.833(0.072)	**0.207(0.091)**
		300	0.820(0.083)	0.873(0.106)	0.233(0.151)
	2	50	0.770(0.068)	0.827(0.167)	0.287(0.154)
		100	0.813(0.095)	0.866(0.154)	0.240(0.181)
		200	**0.833(0.074)**	**0.907(0.064)**	0.240(0.134)
		300	0.810(0.063)	0.900(0.065)	0.280(0.150)

unbalanced. In this tree, almost all subtrees have a leaf node as one of their children. Very few leaf nodes are at the same depth, and there is only a single path to the maximum depth of the tree. This structure results from the sparseness of the dataset. The dataset consists of many attributes but any given problem instance contains only a few of those attributes; at each level the decision tree was able to separate a few files from the remaining group. This resulted in a lopsided tree that has a large height and a relatively small node count.

The number of possible nodes in a tree is exponentially related to its height: $2^{\text{height } + \ 1} - 1$. The search space of a tree producing algorithm greatly increases with tree height. The large search space of the learning classifier system is countered by the fact the each problem instance in the LCS may be matched by more than one rule, while in C4.5 each problem only matches a single leaf node. C4.5 uses a different technique to define its nodes than the LCS does for defining its rules. The LCS is able to adapt its action selection based on rule performance and delete poorly performing rules.

Figure 9.6: Visualization of an unbalanced C4.5 decision tree

A comparison of the C4.5 algorithm to the LCS is presented in Figure 9.7. Training and testing accuracies are shown for each of the three initialization methods as well as the original decision tree results. Each initialization method was tuned for population size, and the figure represents the best testing accuracy obtained for each method.

An important observation is that C4.5 outperformed the LCS on training accuracy, but the reverse was true for testing. This shows that the C4.5 algorithm was able to create a more accurate representation of the training set, but could not generalize as well on the testing set. In malware detection, known examples are useful for benchmarking and training, but they are not a realistic representation of real world data as they have already been classified. Detection of unknown threats is key to a malware detection system, and the testing procedure simulates the system encountering previously unseen files. Although C4.5 may do better on training, it appears to overfit compared to the LCS. Overfitting indicates that the system's learning is specific to the structure of the training set and generalization is not optimal.

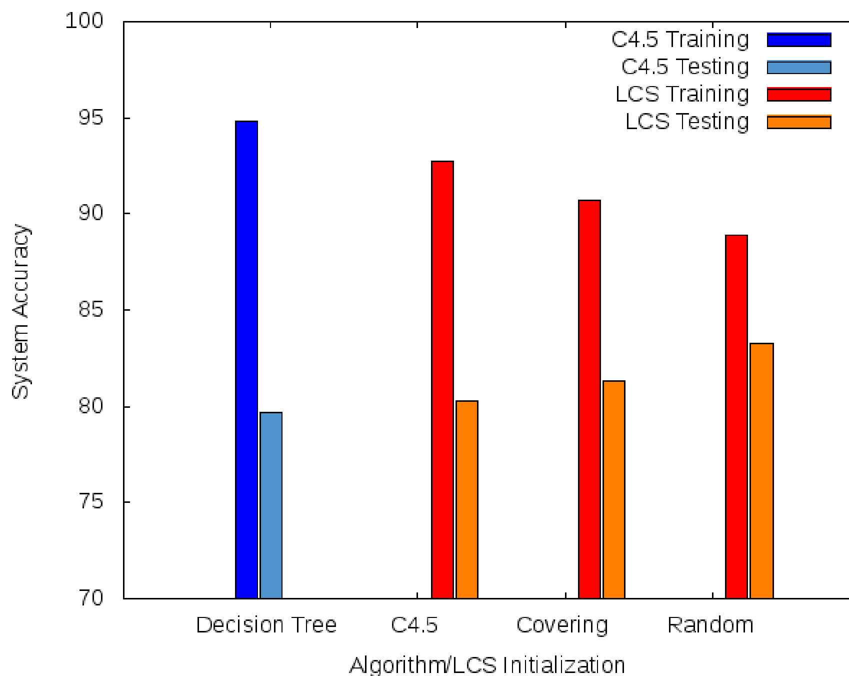


Figure 9.7: C4.5 and LCS comparison

An interesting observation is that as each testing results improved across each method, training accuracy worsened. This is characteristic of overfitting, and is most likely related to the specificity of the rules. C4.5 generates specific rules and trains to a high rate, and the LCS based on those classifications has similar performance, but is slightly more general. The LCS initialized by covering generates rules specifically for training instances, but then generalizes them through mutation and recombination. The initialization method that showed the greatest generalization was random, and it follows from creating the least specific rules. The other methods use the training instances for rule generation and these rules are not as general.

One of the learning classifier system's advantages over C4.5 is its ability to adapt and evolve the population of classifiers. The decision tree is static and cannot adapt to new instances without being re-run over the entire dataset. Visualizing the performance of the system over time, Figure 9.8 shows how the adaptive algorithm starts with an initially poor performance and continuously improves in its ability to classify. Results show that the training performance of the LCS never quite reaches that of C4.5. This helps illustrate the importance of generalization: if an algorithm

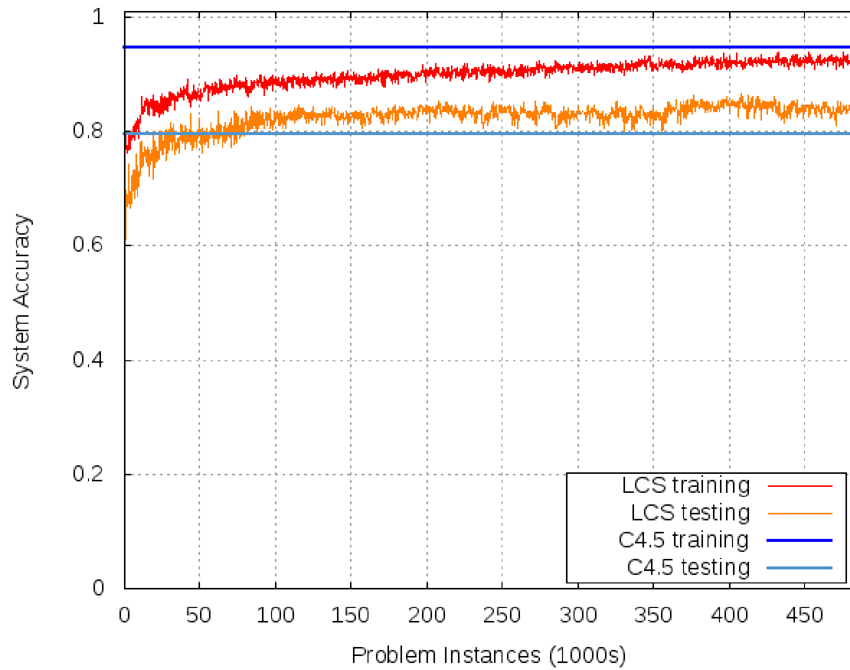


Figure 9.8: LCS evolution compared to C4.5

learns too specifically on the training set, it is said to be overfit, and performance on the test set suffers. An ideal algorithm would be able to train to 100% accuracy and have perfect generalization to any test set.

10. CONCLUSION

This thesis presents an adaptive rule-based malware detection system employing a learning classifier system (LCS). It combines a rule-based expert system with evolutionary algorithm (EA) based reinforcement learning, creating a self-training adaptive malware detection system which dynamically evolves detection rules. The LCS was extended from XCS to use s-expressions for the rule's condition. Promising results are shown, although their accuracy has to be further improved to be competitive with the state-of-the-art. Evidence for the feasibility of using an LCS to evolve malware detection rules is provided. With more features, additional aspects of PE files could be analyzed, which may be expected to enhance detection rates and lower false alarm rates.

The dataset included malware samples from Offensive Computing and non-malicious samples from Windows computers. The set of known malicious and non-malicious files were processed to confirm their maliciousness and features were extracted to be used for rule conditions.

A benchmark for comparing machine learning techniques is presented and the decision tree algorithm C4.5 is used as a performance baseline. Under certain conditions, the system was shown to outperform C4.5 on diverse datasets as well as being better at generalization. Experimental results demonstrate the system's ability to evolve effective rules based on a training set, and its ability to generalize to previously unseen samples contained in a test set. The LCS did not suffer from the feature dimension limitation of C4.5 mentioned in Section 4.5.

Various parameters were compared and the system trained best on multiple datasets with a higher population size, but training performance was dependent on the dataset as well as other problem specifics. No distinct overfitting curve was seen; this is most likely due to a high amount of noise in the dataset. While training and testing accuracies are linked, increasing training performance can negatively affect testing if rules become too specialized.

11. FUTURE WORK

Future work includes using a binary string representation, the more common representation of a condition in an LCS. Bit strings are composed of zeros, ones, and "don't-care" symbols. The s-expressions used in this work allow for other kinds of logical relationships to be expressed, using the logical operator OR. The typical bit-string uses different forms of crossover and mutation, and this would have an affect on evolution. Bit-strings are also trivial to check for subsumption between two rules. If one condition logically subsumes another, it is possible to prevent offspring from being added to the population that are subsumed by experienced individuals already in the population. This operator could be added for s-expressions, but would require significantly more computation time to compare two individuals.

Genetic programming utilizes different genetic operators than EAs employing binary representations, and the subtle differences between the two should be studied. Customizing the system to use fuzzy rules could help the system generalize better, and creating a fuzzy LCS would allow for features to be defined abstractly. A comparison between a Michigan style and a Pittsburgh style LCS would be useful to determine how changing the unit the EA operates on, from a single rule to a ruleset, affects the evolution and performance of the system.

Subtree comparison would reduce the processing time of the system. By evaluating subtrees of individuals, results could be cached on the subtree level instead of on the individual level, and fitness evaluation computation time would be decreased. Subtree comparisons could also be used to reduce tree complexity by eliminating redundant parts of each condition.

Extracting more features from PE files would allow for a better approximation of the problem; it would create an environment more realistic of the real world but would also expand the search space the algorithm is required to search. With more genetic material available, the total number of possible conditions would increase and this would have an effect on how the algorithm performs. An assumption made by this research is that malicious files are identifiable by their extracted features, and the more features available, the more reasonable this assumption becomes.

The current system extracts static features from the files stored on disk, and is a purely static analysis approach. By utilizing dynamic analysis, additional features could be obtained from running the executables. Dynamic analysis could enhance the abilities of the system, but would also require careful environmental setup, as malicious code would be executing.

Additional comparisons to machine learning techniques would provide further insight into how well evolutionary techniques perform versus alternatives. C5.0 has multiple improvements over C4.5 (e.g., shorter run-time, smaller decision trees) and is expected to have better performance than C4.5 [42]. Random forest is another classification technique that builds classification (decision) trees, but this technique is also known to be prone to overfitting [43]. Comparing these algorithms to the LCS on the benchmark malware dataset would show the advantages and disadvantages of different machine learning techniques.

BIBLIOGRAPHY

[1] Jonathan J. Blount, Daniel R. Tauritz, and Samuel A. Mulder. Adaptive Rule-Based Malware Detection Employing Learning Classifier Systems: A Proof of Concept. In *Proceedings of COMPSAC 2011 - the 35th IEEE Computers, Software, and Applications Conference*, 2011.

[2] Annual Report PandaLabs 2010. Technical report, Panda Security, 2010.

[3] The Business of Cybercrime - A Complex Business Model. Technical report, Trend Micro Inc., 2010.

[4] Internet Security Threat Report, Vol. 16. Technical report, Symantec Corp., 2011.

[5] Weidong Cui. *Automating Malware Detection by Inferring Intent.* PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2006.

[6] Robert Moir. Defining Malware: FAQ. `http://technet.microsoft.com/en-us/library/dd632948.aspx`, 2003. [Online; accessed 8-8-2011].

[7] Symantec Report on Rogue Security Software. Technical report, Symantec Corp., 2009.

[8] N. Idika and A.P. Mathur. A Survey of Malware Detection Techniques. *Purdue University*, 2007.

[9] M.Z. Shafiq, S.M. Tabish, and M. Farooq. PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. In *Virus Bulletin Conference (VB), Switzerland*, 2009.

[10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41:15:1–15:58, July 2009.

[11] M.D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[12] Jesse C. Rabek, Roger I. Khazan, Scott M. Lewandowski, and Robert K. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of WORM 2003 - ACM Workshop on Rapid Malcode*, pages 76–82, New York, NY, USA, 2003. ACM.

[13] Keehyung Kim and Byung-Ro Moon. Malware Detection based on Dependency Graph using Hybrid Genetic Algorithm. In *Proceedings of GECCO 2010 - the Genetic and Evolutionary Computation Conference*, pages 1211–1218, New York, NY, USA, 2010. ACM.

[14] M.Z. Shafiq, S.M. Tabish, and M. Farooq. On the Appropriateness of Evolutionary Rule Learning Algorithms for Malware Detection. In *Proceedings of GECCO 2009 - the Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2609–2616. ACM, 2009.

[15] Monu Bambroo. Intrusion Detection using Fuzzy Logic and Evolutionary Algorithm Techniques. Master's thesis, Missouri University of Science and Technology, Rolla, MO, USA, 2005.

[16] Gregory Anthony Harrison and Eric W. Worden. Genetically Programmed Learning Classifier System Description and Results. In *Proceedings of GECCO 2007 - the Genetic and Evolutionary Computation Conference*, pages 2729–2736, New York, NY, USA, 2007. ACM.

[17] Offensive Computing. Community Malicious code research and analysis. `http://www.offensivecomputing.net/`. [Online; accessed 8-8-2011].

[18] Johannes Plachy. The Portable Executable File Format. `http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile.html`. [Online; accessed 8-8-2011].

[19] VirusTotal - Free Online Virus, Malware and URL Scanner. `http://www.virustotal.com`. [Online; accessed 8-8-2011].

[20] Ero Carrera. pefile - a Python module to read and work with PE (Portable Executable) files. `http://www.code.google.com/p/pefile/`. [Online; accessed 8-8-2011].

[21] Craig S. Wright. Packer Analysis Report Debugging and unpacking the NsPack 3.4 and 3.7 packer. Technical report, SANS Institute, August 2010.

[22] T. Brosch and M. Morgenstern. Runtime Packers: The Hidden Problem. *Black Hat USA*, 2006.

[23] C. Shannon and W. Weaver. The Mathematical Theory of Communication. *U. Illinois Press, Urbana, Illinois*, 1949.

[24] R. Lyda and J. Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *Security & Privacy, IEEE*, 5(2):40–45, 2007.

[25] R. Urbanowicz, N. Sinnott-Armstrong, and J. Moore. Random Artificial Incorporation of Noise in a Learning Classifier System Environment. In *Proceedings of GECCO 2011 - the Genetic and Evolutionary Computation Conference*, pages 369–374. ACM, 2011.

[26] J.R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.

[27] K.A. De Jong and W.M. Spears. Learning Concept Classification Rules using Genetic Algorithms. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 651–656, 1991.

[28] Orange - Open-source data mining and machine learning suite. `http://orange.biolab.si/`. [Online; accessed 8-8-2011].

[29] J. H. Holland. Adaptation. In R. Rosen and F.M. Snell, editors, *Progress in Theoretical Biology IV*, pages 263–293. Plenum Press, New York, 1976.

[30] J.H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligenc*. The MIT Press, 1975.

[31] Larry Bull. Learning Classifier Systems: A Brief Introduction. In *Applications of Learning Classifier Systems*, pages 3–14. Springer, 2004.

[32] Ryan J. Urbanowicz and Jason H. Moore. Learning Classifier Systems: A Complete Introduction, Review, and Roadmap. *Journal of Artificial Evolution and Applications*, 2009:1:1–1:25, 2009.

[33] M.V. Butz. *Rule-based evolutionary online learning systems: A principled approach to LCS analysis and design*. Springer Verlag, 2006.

[34] Stephen Frederick Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, Pittsburgh, PA, USA, 1980.

[35] Tim Kovacs. Towards a Theory of Strong Overgeneral Classifiers. In *Foundations of Genetic Algorithms*, pages 165–184. Morgan Kaufmann, 2000.

[36] S.W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary computation*, 2(1):1–18, 1994.

[37] L. Bull and T. Kovacs. Foundations of Learning Classifier Systems: An Introduction. *Foundations of Learning Classifier Systems*, pages 1–17, 2005.

[38] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, 2003.

[39] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.

[40] R. Poli, W.B. Langdon, and N.F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises UK Ltd, 2008.

[41] M.V. Butz, T. Kovacs, P.L. Lanzi, and S.W. Wilson. How XCS Evolves Accurate Classifiers. In *Proceedings of GECCO 2001 - the Genetic and Evolutionary Computation Conference*, pages 927–934, 2001.

[42] R. Quinlan. Data Mining Tools See5 and C5.0. 2004.

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

VITA

Jonathan Joseph Blount was born on September 24, 1987 in Tampa, Florida. After graduating from Park Hill South High School in May of 2005, Jonathan started his undergraduate career at Missouri University of Science and Technology.

After graduating with a Bachelor of Science in computer science and a Bachelor of Science in computer engineering in May 2009, he enrolled as a computer science Master's student at Missouri S&T. During Fall 2009, Jonathan enrolled in a cooperative education program with Sandia National Laboratories. At the end of the co-op, Jonathan was accepted into Sandia's Critical Skills Master's Program. In Spring 2010, Jonathan joined the Trustworthy Systems Laboratory directed by Dr. Miller. In Fall 2010, Jonathan joined the Natural Computation Laboratory directed by Dr. Tauritz. Jonathan completed his degree requirements for his Master of Science degree in Computer Science in August 2011 and will continue his employment at Sandia National Laboratories.