SAND2011-2448C

Adaptive Rule-Based Malware Detection Employing Learning Classifier Systems: A Proof of Concept

Anonymous Anonymous Anonymous

***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 paper is to provide evidence for the potential of learning classifier systems to improve the accuracy of malware detection. This paper presents a proof of concept for adaptive rule-based malware detection employing learning classifier systems, which combine 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. Experimental results are presented which 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.**

***Keywords*-Learning Classifier Systems; Malware Detection**

I. Introduction

Malware is an ever-growing threat to computer systems, and security researchers are competing with malware authors to stay ahead of one another. The number of different strains and types of malicious software has been on the rise for years. A recent report by PandaLabs, a malware research laboratory, stated that "in 2010, cybercriminals 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 [1]. Each year brings an increase in malicious files; the average number of threats created per day has gone from 55,000 in 2009 to 63,000 in 2010 [1]. There is a variety of ways to detect malicious software. Malware detection is the primary step in preventing a computer system from infection, protecting it from potential information loss and system compromise. A majority of anti-virus software uses signature-based techniques that utilize a pre-defined set of signatures [2]. This is a reactive approach: until a signature is created for an exploit, the exploit will elude detection by traditional anti-virus software.

Non-signature based malware detection techniques include dynamic and static analysis. Dynamic analysis, including analyzing machine-level code, static calls from disassembled code, and run-time API calls, suffers from performance overhead and high false alarm rates [2]. Static analysis can be time consuming, especially when malware uses code obfuscation techniques.

The goal of the research presented in this paper is to create a self-training adaptive malware detection system which dynamically evolves detection rules. A learning classifier system (LCS) is a rule-based, expert system with Evolutionary Algorithm (EA) based reinforcement learning [3]. An EA is a "computational search technique which manipulates (evolves) a population of individuals (rules) each representing a potential solution (or piece of a solution) to a given problem" [3]. The LCS responds to inputs from the environment by selecting a response that is expected to maximize environmental reward [4]. The fitness of the classifiers is adjusted based on the environmental reward, which is the reinforcement learning component of the LCS. At set intervals, an EA evolves the set of classifiers, replacing some of the weaker classifiers with newly created classifiers, which comprises the rule discovery component.

The goal of a malware detection system is to identify as much malware as possible while allowing authorized software to execute. To test a malware detection system, known pieces of malware are presented to it, as well as clean files. The set of software used for testing must already be tagged in order to grade how well the system performs. This also provides an idea of how well it will operate in the real world with unknown, zero-day exploits. The testing framework for this malware detection learning classifier system uses known file sets of malicious and clean files. The files are checked by a third party website, VirusTotal [5], which tests files against 43 anti-virus engines. The reinforcement learning

takes place over a training set, and then the evolved rules are tested on a non-overlapping test set. This ensures that the test set is unknown and brand new to the classifiers, but known to the benchmarking system so it can provide an accuracy rating.

The remaining sections are organized as follows: Section II provides background on LCS and malware detection techniques, Section III introduces the methodology of the detection system and how it runs, Section IV explains the experimental setup, Section V presents preliminary results, Section VI summarizes the research, and Section VII details future directions for the work.

II. Background

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 [2]. The authors of this paper first classified a file as packed or non-packed by using a packer detector. Each set is processed separately by a decision tree to determine if it is malicious or not. Different structural models were developed for packed and non-packed executables. To determine if 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. 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.

Little research has been published on applying LCS to malware detection. John Holland created the precursor to the LCS around his Genetic Algorithm, which later became the LCS when it included a reinforcement learning component [3], [6], [7]. 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 and improve existing ones [3]. Each rule in the LCS population has a fitness values associated with it. The iterative updates to rules' fitness drive the LCS learning mechanism; the environment rewards and punishes rules by incrementing and decrementing their fitness, respectively. Reinforcement learning serves two purposes: "(1) to identify classifiers that are useful in obtaining future rewards and (2) to encourage the discovery of better rules" [3].

A comparative study analyzed evolutionary and non-evolutionary rule learning algorithms to evaluate performance differences [8]. 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 categories 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 more than 90%, and 7 out of the 10 algorithms were above 99%. 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.

The research presented in this paper demonstrates promising results for a custom built malware detection system employing LCS on a dataset more representative of the real world. Different configurations are tested and compared to determine which parameters affect the system's ability to evolve high quality classifiers. The test and training sets contained all types of malware, they were not limited to a specific category. Another step this research took is to make sure the files in the sets were actually considered malicious or clean by submitting

them to VirusTotal. Not all samples from the malicious set were actually malicious.

III. Methodology

The environment of an LCS is the source of input data for the LCS. The input data for this research is a collection of files including both malicious and non-malicious binaries. Malicious software samples were obtained from OffensiveComputing [9]. Non-malicious software was obtained from a fresh install of Windows XP as well as from a computer laboratory environment on a university campus computer learning center computer. Clean executables included software created by Microsoft, Adobe, MathWorks, Oracle, other third parties, and open source software. The samples were divided into non-overlapping training and test sets. While real world computer systems (hopefully!) contain more non-malicious software (goodware) than malware, the unique pieces of malware that traverse a network outnumber the unique pieces of goodware. There is a limited number of legitimate Windows executables that will be sent over a network or stored on a computer, but many types of malware propagate and self-modify to avoid detection. The number of unique executables that are malicious is not bounded. The training set used for the LCS composed of 50% malware and 50% goodware. If the training set were to model an actual computer, the amount of goodware would be higher, and if it modeled unique files flowing over a network, malware would represent a larger percentage. The 50% distribution provides a middle ground for the training set. The training set is run through the LCS evolving rules, and the final population is presented with the testing set to generate results.

There are two parts to the malware detection system, a pre-processing component and the LCS, an overview of the process is presented in Figure 1.

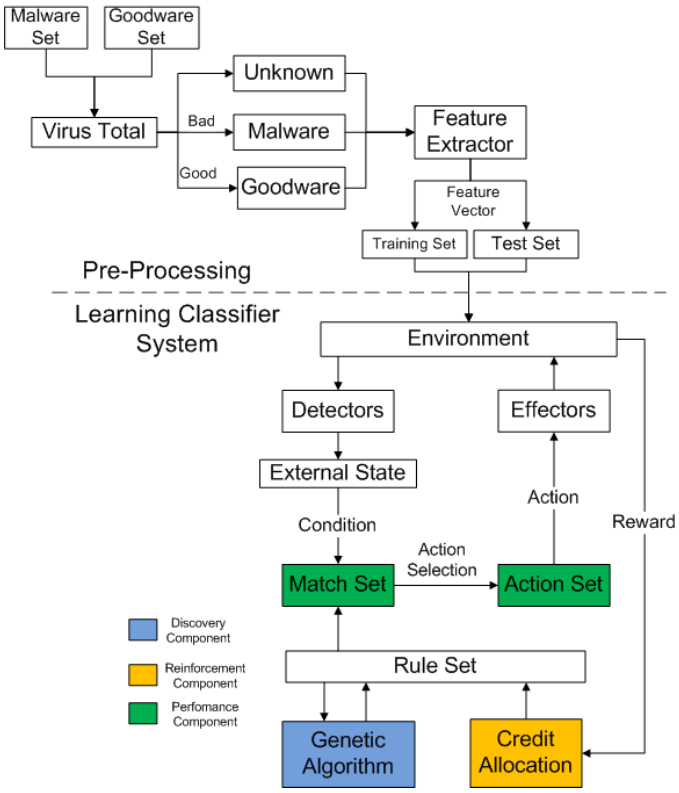

Fig. 1. Malware LCS Diagram

A. Pre-processing

The pre-processing stage analyzes all of the sample files and submits them to VirusTotal [5] for a determination if a file is considered malicious or not. The malicious set of samples from OffensiveComputing.net does not contain just malware. VirusTotal provides a method to create a subset of samples that are known to be malicious or not. If more than 25% of the anti-virus vendors VirusTotal uses to scan a file report malicious, 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 VirusTotal is consistent with the dataset the file came from: i.e., a sample from the malware set not identified as malicious or a sample from the goodware set not identified as 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 system, as they can not be verified as being definitely malware or goodware.

The executable datasets consist of software that runs on the Windows operating system. Windows executables are written in the PE format. PE files start with a DOS header, followed by a PE header, and a number of sections. Each section has a header which describe its data and resources. A typical Windows application has nine predefined sections: .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug [10]. Each section has a different purpose, for example .text is used to store program code, .data is used for global variables. One important section of the PE file is the import data section, .idata. This section 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) that the function is stored in. This research

assumes that malicious files will be distinguishable from goodware based upon the structure of the PE file including the table of imported functions. 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 to improve malware detection. The IAT is used from each file to generate a feature list containing all of the imported functions the executable references. The feature extraction part of the system was implemented using an open source tool called pefile [11].

B. Learning Classifier System

After pre-processing all of the files in the dataset, the LCS randomly initializes a population of rules using the vector of features extracted from malicious files and non-malicious files. The feature vector includes a list of functions imported by the files. The LCS uses an EA to discover new rules from the initial population of randomly generated rules. For initialization, rules were created using a list of the most popular imports from the training set.

1) Environment: The environment of an LCS is the source of input data to the algorithm. The malware LCS environment is divided into a training dataset and a testing dataset. The LCS interacts with the environment through the use of detectors and effectors. Detectors encode the current state of the environment and effectors translate action messages into actions that modify the state of the environment. For the malware detection system, the action that effectors apply to the environment is the decision of whether or not a file is malicious. In a real world system, the operating system would then decide to block a download or stop a file from executing. The environment immediately rewards the system, by checking whether VirusTotal agrees with the action performed and giving a corresponding reward or punishment.

2) Population: The EA in our LCS operates at the level of individual rules, a Michigan-style LCS. The entire population represents a solution. As opposed to a Pittsburgh-style LCS, where the population is made up of rule-sets, where each one is a solution, and the EA operates on the level of an entire rule-set [3]. An individual rule in an LCS has a condition which is encoded in its genome, an action, a fitness value, and a prediction value. Each rule's genome (condition) is a tree structure where internal nodes are one of the logic operators AND, OR, NOT and leaf nodes (terminals) are a single feature. Rules are allowed to grow up to a maximum tree height, in order to keep processing time from growing indefinitely. An example portion of a rule's genome is shown in Figure 2, the subtree's internal nodes are logical operators and leaf nodes are chosen from the set of features. The action for our malware detection rules is to decide whether a file is malicious or not.

For the malware detection system presented in this research, the prediction value of a rule is an accuracy rating based on how the rule has performed so far. For each rule, a record is kept of how many files a rule got correct and incorrect. This record is used to calculate a rule's accuracy and is used by the LCS to choose the action set of rules. During training, if no rules match a given malicious file, a covering operator creates a rule that has a matching condition. This rule is inserted into the population with a chance of spreading its genetic material to offspring, allowing the population to classify the file.

3) Mutation: The system uses random variation to mutate individuals by replacing the subtree starting at a randomly selected node with a randomly generated tree. The height of a child created by mutation can exceed the height of its parent, but it is limited by a $maxheight$ parameter which was adjusted to determine how tree height can affect a rule's performance. Mutation of rules used the same set of imports as initialization, a list of the most popular imported functions.

4) Recombination: Recombination takes places by swapping subtrees between two parents. All rules in the population are considered for recombination, as a panmictic population. A subtree starting at a random node is chosen from each parent, and the subtrees are swapped to create two new children.

The max height of an individual's genome is limited by $maxheight$, and since recombination has the possibility of increasing an individual's height, during recombination the second parent's random node selection was limited to those nodes which would keep the two created children's height below the maximum.

5) Fitness: In an LCS, fitness is determined by the environment, as rewards and punishments are given to rules that were acted upon. The LCS checks whether a feature is malicious or not and adjusts the fitness of individuals who match the feature. Each rule in the population is a parse tree, and can be compared to the extracted features from a file. If the parse tree matches the feature, it is put into the LCS's match set. For every file introduced to the LCS, reward is shared among rules in the match set. If the file is malicious, the rules in the match set are rewarded and if it is non-malicious they

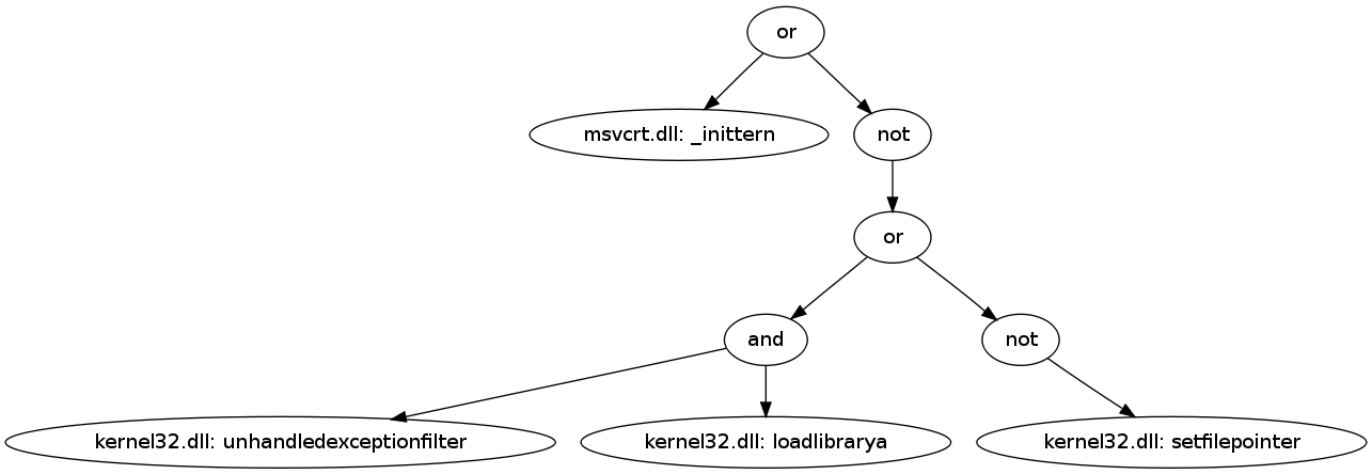

Fig. 2. Visualization of a subtree of a generated rule

are punished.

The fitness function is run for every file presented to the LCS, and the number of files presented before running the EA is a configurable parameter. A random subset of files are chosen from the training set, and presented to the LCS. For each generation, a different random subset is chosen. This ensures file are presented equally and in a random order.

6) Evolutionary Algorithm: The EA is run periodically on the rules to create new rules and eliminate poor performers. The selection mechanism used for parent and survivor selection is tournament selection. A set of classifiers are chosen at random and the one with the highest fitness is chosen to become a parent. The LCS stochastically chooses mutation or recombination to create child rules, which are then put into the population. Every generation of the LCS, the fitness of all individuals is reset, a number of files are presented, and those rules that classify the files correctly are rewarded and incorrect classifiers are punished. An individual's fitness may change each generation depending on which files that are randomly chosen to be presented and the new children that were introduced. This encourages rules to correctly classify a majority of the training files.

After determining the fitness of all individuals, tournament selection is used to choose survivors. Once the the population is trimmed back down to the original size, the genetic algorithm has finished and the LCS continues presenting files from the environment.

As the system evolves rules, each was given an independent fitness which affected whether it was chosen for parent and survivor selection, as well as an accuracy rating based on how it scored malicious and non-malicious files. The overall system also has an accuracy rating. Depending on the rules that make up the match set, the system chooses whether or not to consider a file malicious. If the rules that mark a presented file malicious have a higher accuracy than those that do not, the system treats the files as malicious. Correspondingly, if the set of rules that vote a file is not malicious, the LCS chooses that action. This gives each rule a score, as well as the population as a whole, and as individual rules evolve to better classify executables, the accuracy of the system will improve as well. Various parameter configurations were tested, the results were analyzed to determine which parameters gave the best results. The following sections show how adjusting a single parameter affected the LCS.

IV. Experimental Setup

The goal of this system is to evolve rules that will identify malware based on reinforcement learning. The pre-processing step runs once to extract features from the files used in the experiment and this collection of features is divided into a training and testing set. The LCS evolved rules over the training set, then it evaluated over the testing set. A number of different parameters were adjusted to determine the sensitivity of different configurations.

The main feature that was extracted from the files was the list of imports from the import address table. This limited feature was able to produce promising results by itself, and shows the usefulness of an LCS to enhance malware detection. The number of imports varied per file, and there was a noticeable difference between non-malicious and malicious sets. Figure 3 shows the distri-

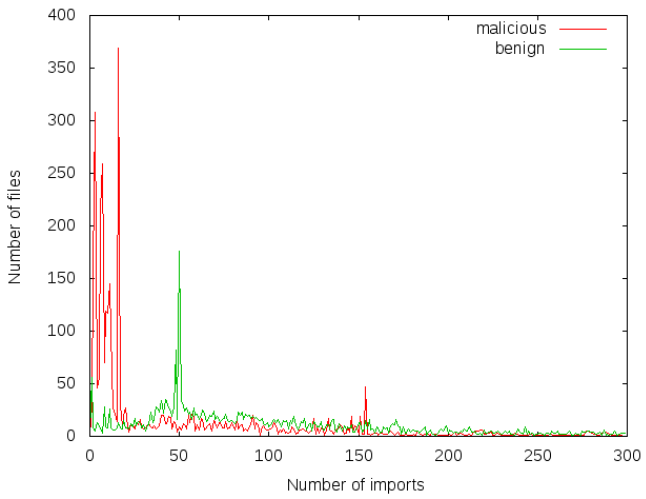

Fig. 3. Distribution of number of imports per file

bution of the number of imports per file for both sets. Malicious files generally have fewer number of imports, with multiple peaks from $1-16$ where non-malicious files peak at 50 imports per file. This is logical as the non-malicious set contains general DLL files, which are shared libraries offering a wide range functionality, and malware is typically written to specifically target one vulnerability.

V. Results

The LCS was trained on a set of malware and goodware and then tested on a non-overlapping set of malware and goodware. A rule's classification accuracy on a file is one of the following four categories:

1) True positive (TP): detects a malicious executable.
2) False negative (FN): does not detect a malicious executable.
3) True negative (TN): does not detect a non-malicious executable.
4) False positive (FP): detects a non-malicious executable.

Three different metrics were tracked: (1) system classification accuracy, (2) system detection rate, and (3) system false positive rate. These are defined mathematically:

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

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

$$\text{false positive rate} = \frac{FP}{FP+TN}$$

Parameter Name	Parameter Value
initialization	uniform random
population size	1000
offspring size	100
operators	['and', 'or', 'not']
operator rate	75%
parent tournament size	5
survivor tournament size	3
crossover vs mutation	75% vs 25%
initial tree height	6
max tree height	10
termination	500 generations

TABLE I
EA strategy parameter table

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.

The list of imports from a file is not a very rich feature set, and there are malicious files which import the same functions as non-malicious files. Any detection system would not be able to tell the difference between files with identical features. The group of such files in this research consisted of 181 malicious files, which were removed from the dataset to increase the usefulness of the rules generated by the LCS. Additional features are needed to improve the accuracy of this system and these are discussed in the future work section.

The list of parameters is presented in Table I. During initialization, each tree is built from the top down. Every node has a chance of becoming a function or terminal node, the operator rate is the chance of the node becoming a function node (operator). If it is, an operator is randomly chosen from the list of operators, and the appropriate number of child nodes are created in the same fashion. If a tree reaches the initial tree height, the node becomes a terminal node.

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

A. Number of top imports

The number of imports used from the feature list to initialize rules is an important parameter that affects how well a rule can classify the set of malware. The list of top imports includes the most popular referenced functions

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

TABLE II
PE ERRORS ENCOUNTERED

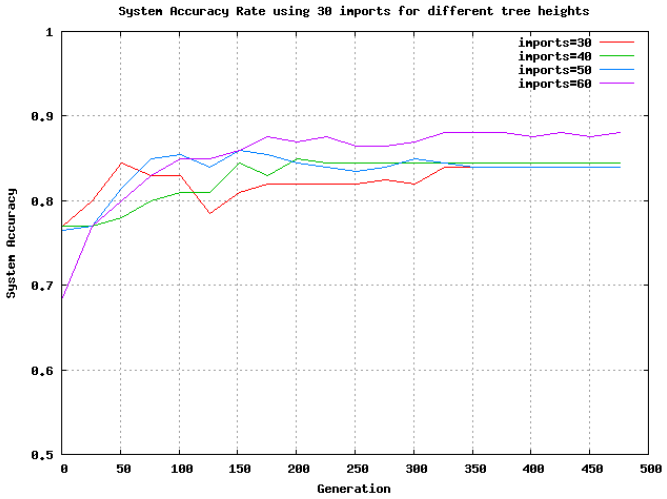

Fig. 4. System Accuracy Rate (200 files, tree height of 6)

from the training set's malicious and non-malicious files. Figure 4 shows the results from running the system using 200 files in the training set and 200 files in the test set and the parameters listed in Table I.

The same test was also run using 300 files in each of the training and test sets, results are shown in Figure 5. Another test was run, increasing the maximum tree height from 6 to 8, shown in Figure 6. The results were similar as the previous trial, the number of imports does not appear to significantly influence the performance of the rules, although the runs where the number of imports was set to 60 did very well in all cases. Allowing the rules to choose from a larger number of imports allowed rules to become more unique and the system accuracy improves. Each individual rule's accuracy affects how it is chosen for parent or survivor selection, but it is the collection of rules as a population that determines how well the system performs.

Figure 7 compares the training accuracy versus the testing accuracy, the training accuracy steadily improves, while the testing accuracy after an initial improvement begins to fluctuate without significant improvement.

The system accuracy, detection rate, and false positive rate (FPR) for the runs are shown in Tables III, IV, V. The best overall accuracy was 88.7% with a detection rate of 90% and FPR of 12.3% and was for the run that had 300 files in each of the test and training sets and a tree height of 8.

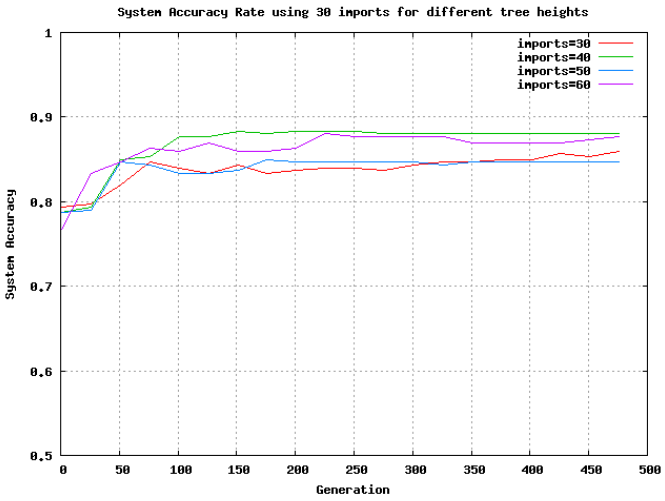

Fig. 5. System Accuracy Rate (300 files, tree height of 6)

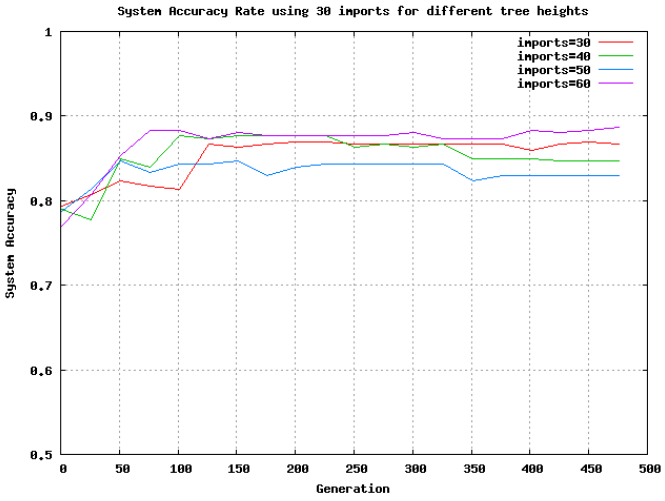

Fig. 6. System Accuracy Rate (300 files, tree height of 8)

Imports	System Accuracy	Detection	FPR
30	0.840	0.840	0.160
40	0.845	0.870	0.171
50	0.840	0.840	0.160
60	0.880	0.850	0.096

TABLE III
EXPERIMENTAL RESULTS (200 FILES, TREE HEIGHT OF 6)

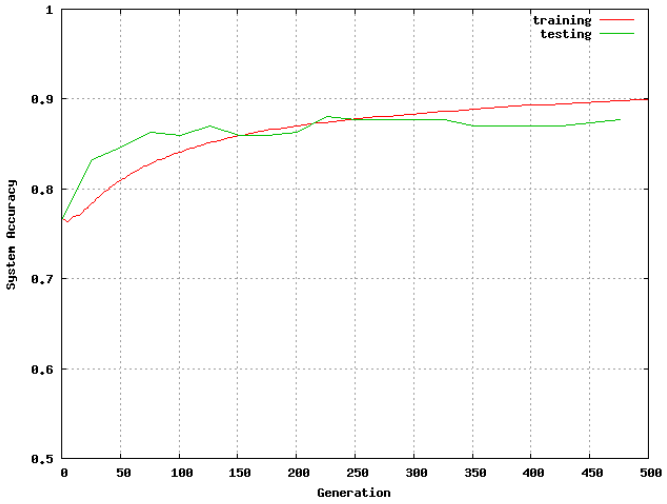

Fig. 7. Training vs. testing accuracy rate

Imports	System Accuracy	Detection	FPR
30	0.860	0.873	0.149
40	0.880	0.873	0.115
50	0.847	0.867	0.167
60	0.877	0.880	0.126

TABLE IV
EXPERIMENTAL RESULTS (300 FILES, TREE HEIGHT OF 6)

Imports	System Accuracy	Detection	FPR
30	0.867	0.893	0.152
40	0.847	0.867	0.167
50	0.830	0.853	0.185
60	0.887	0.900	0.123

TABLE V
EXPERIMENTAL RESULTS (300 FILES, TREE HEIGHT OF 8)

VI. CONCLUSION

This research presented a proof of concept for adaptive rule-based malware detection employing a learning classifier system. It combined a rule-based expert system with EA based reinforcement learning, creating a self-training adaptive malware detection system which dynamically evolves detection rules. The LCS dynamically evolved detection rules which provided promising initial results. While the accuracy is not perfect, anti-virus vendors are constantly battling to stay ahead of malware writers to improve their accuracy rates. A system which uses previous knowledge and learning to develop detection against unknown attacks would be ideal. Evidence for the feasibility of using a LCS to evolve rules is provided. With better features, more aspects of PE files could be analyzed and included in the rules, which would enhance their detection rates and lower their false positive rates.

The results reported in this study used malware samples from the OffensiveComputing website 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 evolution. While the feature set could be expanded, experimental results demonstrate the system's ability to evolve effective rules based on a training set, and generalized to previously unseen samples contained in a test set: this would be representative of a system encountering a zero-day exploit.

VII. FUTURE WORK

Future work for this research includes expanding the feature set to include other aspects of PE files. Besides the import address table, other features to investigate include the name and size of all PE sections, the entropy of sections, and other aspects that can be statically extracted from PE files.

Additional features would allow files with corrupt or missing Import Address Tables to be analyzed. Malware that has corrupted part of its PE file structure may be still be identifiable depending on what features can be extracted from it. Packed files were not given special consideration in this study, but packed files have different structural features than unpacked files. Future work will determine how these features affect the detectability of malicious files.

The current system can not tell the difference between files that import the same list of functions, and this decreased the total size of the dataset the system could run on. By expanding the set of features the LCS uses, files can be better distinguished, and detection rules will evolve to be more accurate.

Benchmarking the system on larger and more diverse datasets will provide additional validation of its potential for malware detection. Tuning of the LCS may be expected to further increase system performance.

REFERENCES

[1] PandaLabs, "Annual Report PandaLabs 2010," Panda Security, Tech. Rep., 2010.

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

[3] R. J. Urbanowicz and J. H. Moore, "Learning classifier systems: a complete introduction, review, and roadmap," *J. Artif. Evol. App.*, vol. 2009, pp. 1:1–1:25, January 2009.

[4] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*. Springer Verlag, 2003.
[5] VirusTotal - Free Online Virus, Malware and URL Scanner. http://www.virustotal.com.
[6] J. Holland, "Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence," 1975.
[7] L. Booker, D. Goldberg, and J. Holland, "Classifier systems and genetic algorithms," *Artificial intelligence*, vol. 40, no. 1-3, pp. 235–282, 1989.
[8] M. Shafiq, S. Tabish, and M. Farooq, "On the appropriateness of evolutionary rule learning algorithms for malware detection," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. ACM, 2009, pp. 2609–2616.
[9] Offensive Computing. Community Malicious code research and analysis. http://www.offensivecomputing.net/.
[10] J. Plachy. The Portable Executable File Format. http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile.html.
[11] E. Carrera. pefile - a Python module to read and work with PE (Portable Executable) files. http://www.code.google.com/p/pefile/.