Applications of Genetic Algorithms to Malware Detection and Creation

Christie Williams

December 16, 2009

Contents

1 Introduction

This paper explores the application of genetic algorithms to a real-life issue, specifically malware. Malware, or "malicious" software such as viruses, worms, Trojans, denial-of-service tools, etc., is becoming an increasingly major issue. As more and more people use the Internet for services such as banking or bill-paying and companies becoming increasingly dependent on Internet communication, the damage that can be done by malware increases significantly. At the same time, malware is becoming increasingly sophisticated, making it both more damaging (compare modern malware that can record a user's every keystroke to early viruses that did little more than replicate themselves uncontrollably) and more difficult to detect (most modern malware is deliberately designed to avoid detection). Faced with these challenges, current methods of protecting against malware, which are often based on static "signatures" updated after the malware is already "live" cannot adequately protect users. The process of creating and releasing signatures for "known" malware does not have a fast enough response time, which leads to frequent "zero-day" vulnerabilities which users are not protected from. One way to remedy this issue is to change the model of protection used by host-based intrusion prevention systems; rather than rely on signatures produced from known malware, the system instead determines whether a process or file is malware based on characteristics that can be observed in real time, such as behavior or common "malicious" patterns in the executable code. This paper examines using genetic algorithms to classify executables as malware or not, based on learned knowledge rather than static signatures.

The paper is primarily a review of previous work done in the area of genetic algorithms as applied to malware detection and creation; it seeks to gain a general feel for what sorts of work is being done in the field and whether there are promising applications or theoretical ideas. In addition, it is an attempt to synthesize an idea of what the trends may be in this area of research, and where further research in the area may lead. The papers presented are from the Association for Computing Machinery (ACM), in particular from various yearly Conferences on Genetic and Evolutionary Computation (GECCO). They were selected based on providing a good survey of the current state of research being done in this area and for representing several different facets of the larger problem.

Section 2 of the paper explores the suitability of genetic algorithms to malware classification, first in Section 2.1 with a paper that compares the performance of a series of genetic algorithm-based classifiers with a series of classifiers that are not based on genetic algorithms. Sections 3 and 4 summarize two papers that present specific ways of applying genetic algorithms to the problem of malware classification; Section 3 deals with using a genetic algorithm to optimize parameter weights used to classify malware in "real time", and Section 4 presents a way of evolving signatures in a manner similar to antibodies in animals in response to detected threats. Finally, Section 5 explores the use of genetic algorithms in evolving malware itself.

Once the papers are summarized, Section 6 explores the connections between the various facets of the problem presented. The first question, addressed in Sections 6.1 and 6.2 is how effective genetic algorithms actually are for classifying malware. This discussion has two main focuses: first, in an overall sense, whether genetic algorithms can be reasonably applied to malware detection as a whole (Section 6.1), and secondly, whether there are certain aspects of malware detection that genetic algorithms are more or less well suited for (Section 6.2). Finally, the issue of using genetic algorithms to evolve malware is addressed in Section 6.3.

1.1 Why Genetic Algorithms and Malware?

This area of research is of interest to me for a couple of reasons. Firstly, I am interested in computer and network security, and malware detection is a large part of this area. There is a huge amount of malware currently in existence, but the biggest problem is often determining whether what is observed is actually dangerous or not. I work part-time at a company that monitors clients' networks for security threats, and as such spend a good share of time investigating "potential" security threats. In many cases, something entirely innocuous will trigger an alert, but the alerting software is not intelligent enough to determine whether it is a threat or simply normal benign data. Instead, a human has to look at the specific problem and determine whether it is a threat or not. Even for a person this is not easy, and judgments often are based primarily on what an analyst has seen previously. To me, it seems that having threat detection software that learns the same sorts of patterns that human analysts pick up would help dramatically with reducing the number of false positives reported and make threat detection more streamlined and useful.

To this end, some sort of machine learning algorithm seems like a logical way of training threat-detection software to recognize malware. I am particularly interested in genetic algorithms as I am fascinated by the way they mimic natural evolutionary processes. To me, the idea of using simple elimination based on fitness to train the algorithm is more interesting than some of the more theoretical, math-based approaches of other machine-learning methods and I would like to explore it further. In addition, genetic algorithms are supposed to work better than many other types on real time data, which is a decided advantage when running malware detection on a "live" system.

2 Effectiveness of Genetic Algorithms for Detecting Malware

The first paper analyzed is from 2009 by Shafiq, Tabish, and Farooq, and deals with a comparison of ten different rule-based algorithms [4]. Five of the algorithms are evolutionary (specifically XCS, UCS, GAssist-ADI, GAssist-Intervalar, and SLAVE) and five are non-evolutionary (specifically RIPPER, SLIPPER, PART, C4.5 rules, and RIONA) [4]. These ten algorithms were all applied to the classification problem of detecting malicious executables, and performance measures including classification accuracy, the number of rules generated, "comprehensibility" of the rules, and processing overhead were recorded for each [4].

2.1 Why Examine Genetic Algorithms?

According to the paper, evolutionary rule-learning algorithms have been receiving significant attention recently, and when tested on various benchmarks do relatively well (in some cases on par with non-evolutionary algorithms). In general, they appear to be a promising set of classifiers, and are an area of current interest. However, most of the research has not looked at performance measures beyond classification accuracy, which overlooks other significant issues such as processing overhead. This becomes increasingly important when dealing with "live" systems such as malware classification, as any system that bogs down too much is not one that can be used effectively in real-world applications. Other factors such as the number of rules and the comprehensibility of those rules are also important in a real-world setting as they effect how useful the classifier is to an end-user attempting to use it for a practical application. One prime example in malware detection is how easy to read and understand (referred to in the paper as "comprehensibility") the rules produced by the algorithm are. If they are too obscure, they are of little use to the "malware forensics expert" using the tool, making the algorithm a less desirable choice [4]. This paper is an attempt to analyze the relative merits of evolutionary and non-evolutionary rule-learning algorithms to determine whether the evolutionary versions stack up well against their non-evolutionary counterparts in the above-listed performance measures.

2.2 Methodology

For this investigation, ten classifiers, five evolutionary and five non-evolutionary, were applied to a dataset of malicious and non-malicious executables for the Microsoft Windows operating system. The evolutionary algorithms were representative of three main types of such algorithms: two (XCS and UCS) are "Michigan"-style classifiers, two (GAssist-ADI and GAssist-Intervalar) are "Pittsburgh"-style classifiers, and one (SLAVE) is a genetic fuzzy classifier. Similarly, the non-evolutionary algorithms fall into three common distinct types as well: three (RIPPER, SLIPPER, and PART) are iterative rule learners, one (C4.5 rules) is a decision tree based rule learner, and one (RIONA) is an instance based rule learner [4]. The executable samples were obtained from the local area network of the authors' lab (benign executables) and from the 'VX Heavens Virus Collection', a publicly-available malware database [4]. The total dataset contained 11,786 executable files, from which 1,447 were benign and 10,339 were malicious [4]. From these samples, 189 features were extracted and used to classify executables [4]. The malicious executables in the training set were divided into 8 categories of common malware types (such as backdoors/sniffers, Trojans, viruses, DoS/Nukers, etc.) in order to increase the number of samples in each category [4]. Each of the classification algorithms was applied to the malware dataset, and the four previously listed performance measures (classification accuracy, rules generated, comprehensibility, and processing overhead) were recorded.

The first performance measure, classification accuracy, was represented in several ways. The primary measure, and the one used to make comparisons, was defined as

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

where TP represents the number of true positives, TN the number of true negatives, FP the number of false positives (benign classified as malicious), and FN the number of false negatives (malicious classified as benign)[4]. This represents the percentage of correct classifications out of all classifications made. Other secondary measures of accuracy used were "gain ratio" (which represents the classification power of the features and can show which features are more or less meaningful when doing classification) and "symmetric uncertainty" (the difficulty of classifying a given group, i.e., the "Trojan" group)[4]. Further explanation of these secondary measures and graphs of the results can be found in the original paper[4]. These were used primarily to gain a better understanding of the dataset and to determine what aspects of the data were more and less meaningful to successful classification.

The next performance measure reported was the number of rules generated. In general, a larger set of rules means greater complexity and overhead, and the number generated is heavily dependent on the algorithm in question. For example, several of the genetic algorithms specifically limit the number of rules, whereas none of the non-evolutionary ones do[4]. A somewhat related performance measure is the comprehensibility of the generated rules. This measure is subjective, and defined loosely in this paper as `high`, `medium`, or `low`[4]. In general, a rule is considered more comprehensible if it is more "legible" (i.e., how long a given rule is and how many attributes are listed in a given rule) and/or if the set of rules is smaller. In addition, complexities such as rule weights and required rule ordering cause comprehensibility to be rated as lower[4].

The final performance measure used is the processing overhead. This is significant in that it can determine whether an algorithm is feasible to use in real life applications (such as a real time malware scanner). This overhead comprises both the training and the testing overhead (represented as execution time in seconds), as both are important from a practical standpoint[4].

2.3 Results

Overall, the results presented in this paper indicate that evolutionary rule learning algorithms are not a feasible method of classifying malware. They suffer from two main shortcomings: lower accuracy compared to non-evolutionary rule learning algorithms, and are significantly slower (to the point where all but the UCS algorithm are infeasible in a real world scenario). In the accuracy measurement, the worst classifier (XCS) had an average accuracy (over all malware types) of 0.9550, while the best (GAssist-ADI) had an average accuracy of 0.9917[4]. While these numbers seem quite good, compared to the non-evolutionary algorithms (which all scored 0.9899 or higher, with all but one at 0.995 or higher[4]) they are not as impressive.

Even based on classification accuracy alone, non-evolutionary algorithms come up the winner, but when processing overhead is thrown in, all but one of the evolutionary algorithms is immediately rendered useless. This is because the two Pittsburgh-style classifiers take on average 1.1772 seconds (GAssist-ADI) and 4,060.9 seconds (GAssist-Intervalar) to complete, and two others take over 100 seconds (109.0 for XCS and 775.1 for SLAVE). Only UCS completes in a reasonable time of 16.3 seconds. Comparatively, all but one of the non-evolutionary algorithms complete in well under 100 seconds; the only exception is RIONA which takes 2680.5 on average[4]. When execution time is running in the minutes, the algorithm simply is not feasible for real world applications.

In terms of the number and complexity of rules, the distinctions between evolutionary and non-evolutionary algorithms is less clear-cut. XCS and UCS (evolutionary) both generate close to 10,000 rules, but the rest are all significantly less. The only other algorithm that generates double digits of rules (57) is SLIPPER (non-evolutionary); the rest have fewer than 10[4]. As far as comprehensibility, GAssist-ADI is classified as having a high comprehensibility, as are RIPPER, SLIPPER, PART, and C4.5 rules. All of the rest (besides RIONA, which cannot be classified due to a different representation of rules) are classified as medium[4]. In general, the non-evolutionary algorithms are ranked as somewhat more comprehensible, but this is subjective and not as important to overall usefulness as other performance measures.

One important note regarding the results of this paper must be made. These experiments examine a subset of genetic algorithm-based malware classifiers, rule based learners. While evolutionary algorithms are clearly inferior in this subset, this is not necessarily the case for all algorithms. In fact, other work cited by the authors[4] implies that for other types of algorithms, the evolutionary ones are generally on par with if not superior to their non-evolutionary counterparts. As such, the results of this paper should not be generalized to claim that genetic algorithms as a whole are inferior, though they clearly are for rule based learning approaches.

3 Using Genetic Algorithms to Tune a Real Time Malware Detector

The second paper is also from 2009, by Mehdi, Tanwani, and Farooq, and presents a method for using a genetic algorithm in order to tune the behavior of an In-execution Malware Analysis and Detection scheme (IMAD) [2]. IMAD is designed as a replacement for the traditional static signature-based malware detection common in commercial products today and operates by analyzing the function calls made by an executable at run-time. By analyzing which calls are made in which order, IMAD is able to determine relatively reliably whether a given executable is malware or benign. The genetic algorithm aspect of IMAD is the way in which the "weight" of each parameter used in the classification is decided. Each parameter used in IMAD is represented by a gene in the genetic algorithm, and fitness is determined by the classification success of the chromosome.

3.1 Incentives for a Real Time Classifier

As malware has gotten more and more sophisticated and prevalent, the so-called "zero-day" threat has grown significantly. This is the space of time where a known threat (such as an unpatched vulnerability or new form of malware) is present but mitigating steps (such as a relevant malware signature) have not yet been released. During this time, users are vulnerable to the malware. Current systems generally rely on the release of static signatures, but this takes time and, as malware volume continues to grow, becomes increasingly infeasible. To remedy this, attention has turned to behavior-based detection of malware. This removes the need to have a signature for each piece of malware, and can also allow for more useful remedies such as stopping a malware executable while it is running, rather than detecting it after the damage has been done [2].

Essential to the idea of real-time detection and mitigation are the requirements for high accuracy (many false negatives make the system useless and many false positives make the system potentially unusable due to user annoyance), low processing overhead, and ideally, the ability to stop the execution of malware while it is running. One other aspect of runtime detection over signature-based detection is that it should cut down on problems with obfuscated malware as it is much more difficult to disguise malware actions than the malware code itself. The scheme detailed in this paper seeks to not only provide a feasible non-signature-based detection method, but also to detect malware as soon as possible while it is running and stop its execution [2].

3.2 Optimizing the Classifier

The IMAD scheme is made up of several components, which as a whole log the system calls a process makes, analyze the pattern of the calls, evaluate the "goodness" of the particular pattern, and then classify it as malware or benign. If it is determined to be malware, execution is halted, and if it is determined to be benign, scanning of that process is ended. The genetic algorithm piece of this architecture is the optimization of the goodness values for each feature, along with a few other parameters used in classification. This allows IMAD to learn which features tend to be associated with malware as opposed to with benign programs and classify accordingly [2]. The dataset used for the experiments in this paper is a combination of 100 publicly available pieces of Linux malware in the 'VX Heavens' repository and 180 benign processes gathered from the `/bin`, `/sbin`, and `/usr/bin` directories of Linux [2].

Features in IMAD are referred to as "n-grams", which are series of function calls of length `n`. In theory, with the right value of `n`, sets of function calls will emerge that can be used to reliably predict whether an executable is malware or not. The experiments presented in this paper use `n` values of 4 and 6, which lead to 952 and 1370 n-grams, respectively. In addition, there are three other parameters to be tuned: a smoothing function which controls the weighting of the goodness of previous n-grams by the formula

$$I \leftarrow G + \alpha I \tag{2}$$

(where α is the smoothing value), and the upper bound (above which a process is deemed benign) and lower bound (below which a process is deemed malicious) [2]. All of these become inputs to the genetic algorithm, which treats each input as a gene and optimizes for a chromosome with the highest classification accuracy. This fitness value is calculated with the formula

$$F = \frac{P}{TP} + \frac{N}{TN} + \frac{FP}{P} + \frac{FN}{N} \tag{3}$$

where P is the number of positives (malware), N is the number of negatives (benign), TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives (benign classified as malicious), and FN is the number of false negatives (malicious classified as benign) [2]. The reason the fitness function is so complicated (rather than a simple percent correct) is to weight against the genetic algorithm learning to classify processes at random. This works because such a behavior would lead to a very high F (which is very undesirable as the optimum fitness value is a very small one). In addition, this fitness function tends to weight against premature classification, instead favoring a process remaining unclassified to one that is classified incorrectly (as an incorrect value will penalize two fractions but an unclassified one will penalize only one) [2].

The experiments presented in this paper explore performance both in terms of detection accuracy (over the entire execution of test programs) and the percentage of the suspect program completed before being detected as malware. The first measure is important to determine whether IMAD is detecting malware and benign programs correctly, and the second to evaluate how well IMAD does at detecting and stopping threats at runtime. The authors used 10-fold cross validation to determine the accuracies of both the IMAD algorithm and four other common classification algorithms (a support vector machine algorithm called Sequential Minimal Optimization (SMO), a decision tree algorithm called C4.5, a propositional rule learner called RIPPER, and a naive Bayes algorithm) [2]. Detection accuracy was calculated for all five algorithms using both 4-gram and 6-gram datasets, and the number of false positives, false negatives, true positives, and true negatives was also calculated for each. In addition, the breakdown of malicious, benign, and unclassified executables was determined for IMAD on both 4-gram and 6-gram datasets [2]. The second experiment involved determining how quickly IMAD could detect malicious code during execution. To test this, the authors measured how many n-grams had occurred before IMAD made a malware classification, and normalized this based on the total number of n-grams in the executable [2].

3.3 The Malware Detector in Action

For the most part, IMAD comes up the winner in comparisons between it and the other four classification schemes it was compared with. Overall detection accuracy is not the highest (though it is tied for most accurate when a 6-gram dataset is used), but it has a significantly lower false alarm rate than all competitors. When tested on a 4-gram dataset, IMAD has a total detection accuracy of 75%, higher than both the support-vector machine and naive Bayes (both at 64%), but lower than C4.5 and RIPPER (at 85% and 79%, respectively). On the 6-gram dataset, IMAD did better at 77% (tied with RIPPER), while the rest scored worse (with the lowest being naive Bayes at 58%) [2]. However, the most impressive part of the comparisons presented is the false alarm (false positive) rate for IMAD. On the 4-gram dataset, it had a 3.89% false alarm rate, while the next highest was the SVM at 7.22% and the highest were around 14-15% (C4.5 and RIPPER). On the 6-gram dataset, IMAD did even better with a 0% error rate compared to the next lowest (SVM) at 9.44% and others at around the 15% level [2]. This is a significant plus for IMAD, as even a few false positives significantly lower the usability of a malware detector. Particularly for the 6-gram dataset, overall detection accuracy is on par with other classification methods, and the false alarm rate is significantly better.

One interesting aspect of IMAD is the way it handles samples it is “unsure” about. If the value returned is never outside of the two boundaries, IMAD will not classify it, instead pushing a decision off until more information is known. This can lead to it never classifying a sample, leaving it as undecided even after execution finishes [2]. Functionally, this is the same as considering it benign, as no alarm is raised. When tested on a 4-gram dataset, IMAD leaves around 12% of the samples unclassified (relatively similar percentages from the benign and malicious categories, though a little higher on the malicious side). When tested on a 6-gram dataset, this increases to 18.21%, again with a relatively consistent number of samples from the benign and malicious categories [2]. This is the downside of the very low misclassification values; rather than misclassify samples, IMAD simply does not classify them at all. A slightly fairer approach might be to include non-classified malicious samples as errors, since functionally they are treated as benign.

Finally, the results of in-execution detection time reveal that IMAD does a fairly decent job of detecting malware early on in execution. On the 4-gram dataset, IMAD was able to classify around a third of the malware samples after 20% execution, and this rose to close to 50% on the 6-gram dataset. Around 65% were correctly classified by 50% execution on the 4-gram dataset, which rose to around 75% for the 6-gram dataset. In both cases, only 15-20% of malware took longer than 70% of the execution to be detected [2]. Graphical representations of the percent of malware detected by execution percentage can be seen in the original paper, along with histograms showing the number of malware samples classified at each execution percentage [2]. One note the authors make is that malware damage may not be evenly spread across execution time. For example, early code may deal more with evasion than damage, and conversely some malware may deliver its "payload" early and fail to be detected until later in its execution [2]. Overall, however, IMAD does appear to detect a significant portion of malware well before its execution has completed, which is still an improvement over non-runtime analysis tools.

4 Using Genetic Algorithms to Evolve Malware Signatures

The next paper analyzed is from 2006 and written by Edge, Lamont, and Raines [1]. This paper advances the idea of using a genetic algorithm to evolve signatures to detect malware in much the same way that the human body develops antibodies for diseases. The specific improvement made in this paper is the idea of adding "memory" to the system such that local minima are easier to escape. When a minimum is encountered, the algorithm can revert to an earlier "known good" state and make different choices in order to avoid becoming stuck. This "memory" feature is based on the function of RNA in the human immune system, and leads to the naming of the system the Retrovirus Algorithm, or REALGO. This method differs from that presented in the previous paper in that, rather than using a genetic algorithm to tune a classification system, the genetic algorithm is actually driving the classifier by creating the rules that the classifier uses to detect malware.

4.1 Genetic Algorithm to Mimic Antibodies

Currently, the majority of malware detection systems rely on signatures created based on known malware and pushed out to users. This creates a gap where malware can do damage to users who are not yet protected from it as the threat cannot be detected and mitigated. In addition, with more and more strains of malware, the signature collections are growing at an exponential pace and will not be able to continue to scale well. One way to remedy this is to develop a system that can combine knowledge of past malware with predictions of future malware in order to protect against threats as they emerge. One promising area of research that does this is that of artificial immune systems, in which a computer can initially be "immunized" with a series of signatures, and then continue to evolve new ones based on emergent threats. In addition, computers can share their learned knowledge with other computers, meaning that once one host has seen a piece of malware, it can communicate an effective "antibody" with other hosts to spread resistance to it [1].

One issue with any sort of optimization problem is that of local minima. It is quite common for algorithms to find a local "best" location that is not the global best, but fail to continue to explore looking for a better one. To remedy this, this paper proposes the idea of using a "memory" feature based on RNA in the human immune system. This "memory" allows the algorithm to revert to a known "good" state should the search stagnate, and search in a different "direction" in order to potentially find a better optimum. The authors hypothesize that this addition will help artificial immune systems to be both more efficient and more effective [1].

4.2 Methodology

The REALGO system is modeled heavily on the human immune system, with most of its parts and behaviors designed to mimic natural immune system behavior. Initially, a random "seed" set of antibodies is generated, and the system is then exposed to a series of "antigens" (known malware signatures) so that it can learn what antibodies best detect the malware. Fitness is measured by a representation of "distance" between the bit string in the antibody and the bit string in the antigen (in essence, and antibody is trained to mimic the structure of the antigen signature). These antibodies are evolved using a genetic algorithm, and random mutation is encouraged in order to provide the ability for the antigens to match new malware as well as that in the training set. Further, once the system is running and detects a new piece of malware, it can send the antibody it has learned from this malware to other hosts so that they gain immunity to it as well. If an antibody is unused for too long, it is allowed to "die" in order to keep the number of antibodies from building up to an unmanageable level. On the other hand, an antibody that is matched often can be used to trigger a more rapid response analogous to the "secondary response" the human immune system exhibits under similar conditions [1].

Another key aspect of the way the REALGO is trained is that learning must account for both "positive selection" (the ability to detect malware) and "negative selection" (the ability to not detect benign data, such as parts of the system, as malware). The positive selection training is done as described above, using the genetic algorithm to train against known signatures. Once this is completed, however, the created antibodies must be matched against known good processes and files on the system, and any antibody that matches such a process or file within a certain threshold is removed. In order for the genetic algorithm to continue to function properly, the antibody set must be of a certain size; if too many antibodies are eliminated, more are randomly generated, and the process is repeated. On the other hand, if too many antibodies are created, the ones with the lowest fitness values are dropped [1].

The full algorithm to describe the use of an RNA-like "memory" structure is presented in the paper; here a simple version is provided. For full details, see the original paper [1]. First, the initial antibodies are generated, and evaluated against the training signatures. If, after this step, the antibodies do not match the antigens with a high enough value, an RNA structure (containing a series of antibodies) is imported, and low-scoring antibodies are replaced with higher scoring ones present in the RNA with a certain probability. Flags are available to protect "desired" antibodies (for example, ones which have tended to score highly in the past); if these flags are set, a given antibody will not be replaced. After this, new antibodies are created by "mating" existing antibodies, and the resulting children are evaluated. Any antibodies that fail to match the training set within a certain threshold are culled; if they were mutated from a previous RNA string, they are reverted to that string in order to "reset" them to a known good point. At the same time, any particularly well-scoring antibodies are used to create new RNA strings in order to preserve their "knowledge". The next generation of antibodies is selected to be the `n` best scoring antibodies, where `n` is the desired number of antibodies, and the process continues until all match the antigens within a given threshold [1].

Once the first part of the algorithm has terminated, the resultant antibodies must undergo negative selection to make sure that the antibodies do not detect the system itself as a threat. If this removes too many antibodies, more must be randomly generated and the cycle repeated until enough pass both types of selection. Once this is complete, the system is ready to begin detecting malware. Any time a process or file is accessed, the REALGO compares it to the antibodies. If there is a match within a certain threshold, the system flags it as a virus. Initially, antibodies are "immature" and must request user confirmation regarding whether a process or file is actually a virus; upon receiving confirmation, they become "mature" and can autonomously detect future malware. A "mature" antibody is thus equivalent to the "secondary response" mentioned earlier as it can more efficiently detect and mitigate malware it has seen before. In addition, upon confirmation, this antibody can be distributed to other hosts to spread immunity to the detected threat. The REALGO system also continues to evolve antibodies during runtime, following similar rules to those implemented in the training phase [1].

The authors then evaluate the REALGO system in a couple of ways. First, as there are not good malware search space models, the authors instead test the REALGO against a series of 8 varied benchmark landscapes as a proof of concept on the theory that at least some will bear some resemblance to a good malware detection landscape. Results for each landscape were averaged over 50 runs and the results compared against standard benchmark results for each test function. Parameters were tuned empirically based in part on prior knowledge of the problem domain (for example, a simulated annealing constant was adjusted based on a known tendency of the landscape to contain many separated local minima). Once this testing was complete, the authors then began testing with a small sample set using 5-fold cross validation to gain measures of detection and false positive rates [1].

4.3 Performance of the Malware Detector

In initial testing, the REALGO produces results that are either comparable or superior to other methods such as the Fast Evolutionary Strategies algorithm and the Classic Evolutionary Strategies algorithm. For three benchmarks, the REALGO was significantly better than the FES algorithm, and on the rest it was statistically equivalent. For two benchmarks, the REALGO was statistically superior to the CES algorithm, and it was statistically equivalent on three others. The authors believe that the reason CES performed better on some of the benchmarks is that they were simpler, and thus the lower overhead of CES beat out the REALGO. On more complex benchmarks, however, this was not the case. The full results can be seen in Tables 2 and 3 in the original paper [1].

The authors also evaluated the particular feature that makes the REALGO unique, namely its RNA "memory" structure. Results of student t-tests for the REALGO with and without the RNA enabled were compared; for several of the benchmarks, there is a clear difference between both the REALGO with RNA and without, as well as REALGO with RNA and the FES and CES algorithms. This supports the hypothesis that the addition of a memory function allows the REALGO to optimize a better set of antibodies and thus be more efficient at detecting malware than similar systems without it. Full results of these tests can be seen in Table 4 in the original paper [1]. While these results are still preliminary, they do overall point towards the REALGO being an improvement in the area of artificial immune systems, as well as implying that the concept of evolving malware signatures is a valid way of detecting malware without the standard static signature approach.

5 Evolving Malware Using Genetic Algorithms

The final paper discussed is from 2009, by Noreen, Murtaza, Shafiq, and Farooq, and deals with the idea of using a genetic algorithm to evolve malware rather than a malware detector [3]. This scheme works on the idea that malware (or any other type of program) has a "genotype" representation based on its machine code, and that this genotype can be evolved using a genetic algorithm to produce related but unique versions of the code. This paper used the `Bagle` virus as a proof-of-concept test that unique strains of the virus could be generated from an initial sample. While the proof-of-concept code is a virus, the authors point out that this same technique can be applied to any type of code, not simply malware.

5.1 Why Evolve Malware?

Over the last couple of decades, malware has become an increasing problem. Starting with simple viruses that simply attached themselves to benign executables, they have since progressed to employing a wide range of techniques to defend themselves and evade detection. The most recent malware takes this a step further by changing their code every time they are replicated. This has led to the frequent claim that malware is "evolving", but this is not exactly the case. Rather, the authors claim that this is not true evolution as only the structure, not the functionality, is changed. Thus, this paper seeks to determine whether it is possible to create truly evolving malware, in the sense that it changes the "behavior, functionality, or semantics of a given malware, rather than a mere change in its structure" [3]. Another aspect of the paper, though not an explicitly stated goal, is that it tests whether a genetic algorithm can be used to drive this evolution. This is in contrast to many "malware creation engines" which, while creating variants of a particular virus, do so in a relatively naive manner rather than in a true evolutionary sense.

5.2 Methods Used

For this experiment, the authors built an "evolvable malware framework" consisting of three parts. First, a high-level abstraction (or "genotype") of the malware is created. This can include various parameters such as ports used in an attack, target applications, registry entries created or modified, etc. In short, it defines the various parameters in the malware that can be varied between generated malware versions. Secondly, a genetic algorithm is applied to this high-level representation. For this experiment, the authors tested several different selection and crossover methods and compared the results. Once new malware has been generated, the third step is to translate it back down into machine code so that it can be executed. Once this has been done, the generated malware is tested with commercial antivirus programs in order to determine if it is a known variant of malware. The malware is also executed on a test machine to aid in determining whether it is "really" malware. In order to make the experiment more quantifiable, the scope was restricted to known variants of the "parent" virus only [3].

The particular piece of malware used for this experiment is the `Bagel` virus, "backdoor" virus which has a number of known variants. For this study, the authors selected 15 known versions of the virus, available from malware sources such as 'VX Heavens' and 'Offensive Computing'. The set was divided into training and testing sets; the training set was used to guide evolution of the new viruses, and the testing set was used to compare to see if any of the generated viruses matched "known" strains that had not been used to guide the evolution of the output viruses. The abstract representation of the `Bagel` virus was created based on several of the features that are known to vary between known versions of the virus, including the concealing application, the port number, attachment extensions, virus name, email body, email subject, etc. (a full list with explanations can be found in Table 1 of the original paper) [3]. The fitness function for the genetic algorithm is based on the resemblance between the resultant offspring and the viruses in the training set (a closer match is a more fit offspring). In addition, the offspring are compared to the testing set and the results recorded, though this is not used to guide the evolutionary process. The various genetic algorithm methods that were compared to determine which was the best were split into two categories; the selection method could be one of the roulette-wheel, rank, and tournament methods, and the crossover methods could be one of the one point, two point, and uniform crossover methods. In all cases, the crossover probability was set at 0.75, the mutation probability at 0.005, and the population size at 500. One note about the application of a genetic algorithm in this study: rather than being used in an optimization sense, it is used primarily to generate variation in controlled directions [3].

5.3 Malware Evolution in Action

The results of this experiment clearly show that malware can be successfully evolved using a genetic algorithm. Interestingly, however, the methods of crossover and selection do not appear to be related to the fitness of the offspring produced. Unsurprisingly, the fitness of the offspring produced is higher on the training dataset than the testing dataset, though as noted above, "fitness" is a somewhat arbitrary concept in this case and means primarily that the resultant offspring are more similar to the training dataset. Another item of note is that as the population size increases, the percentage of offspring matching a `Bagel` virus in the training set increases. More interesting is the fact that around 50% of the offspring match virus strains in the testing set. What this effectively means is that the offspring created are thus matches to "unknown" (to the algorithm) viruses which are in fact verified strains of the `Bagel` virus. This means that the algorithm is successfully generating new viruses via evolution. In addition, some of the offspring are detected under other virus names by commercial antivirus software, implying that the genetic algorithm may have evolved other similar virus types as well. Detailed tables showing all of these results can be found in the original paper (Tables 2 and 3, with graphical representations in Figure 9) [3].

6 Putting it All Together: Genetic Algorithms and Malware

Clearly, there is a huge variation in the ways that genetic algorithms can be used to both classify and create malware. In part, this is because genetic algorithms are a technique for optimization rather than a specific method for creating results. This means that they can be applied directly, such as to evolve malware signatures, or indirectly, to aid in optimizing other types of classification schemes. How the genetic algorithm is used, and what it is used for, has at least as much effect on effectiveness as does the problem (detecting malware). That said, however, overall it appears that genetic algorithms do have many promising applications to detecting malware, as well as for creating malware that is better able to evade existing detection systems. In essence, both applications are two specializations of the same problem: generating new variations of a data representation (be it a malware signature or a piece of malware) autonomously and in a manner designed to optimize some goal (such as detection rate or evasion rate). The following sections analyze how well genetic algorithms apply to malware detection in general, specific areas where genetic algorithms shine, and, on the flip side, how genetic algorithms can also be used to build more dangerous malware.

6.1 How Useful Are Genetic Algorithms to Malware Detection?

Despite the overall promise of genetic algorithms to this area of research, the results on whether genetic algorithms are effective to malware detection are mixed. In some instances, they do well (for example, the malware detection schemes presented in both the IMAD [2]) and REALGO [1]) papers do better than other common algorithms on the same problem), but in others, evolutionary solutions are clearly inferior. The first paper analyzed [4]), which compares ten evolutionary and non-evolutionary rule learning algorithms clearly showed that, at least for that type of algorithm, genetic algorithms are vastly inferior. The performance of the evolutionary algorithms was not bad (still around 98-99% for the most part) but could not compare with the non-evolutionary versions. In addition, while 98% accuracy may seem high, for an application such as malware detection this is deceptive. Even a single missed piece of malware can be incredibly damaging, and if the detector presents too many false positives, users will become fed up with it. This is relevant to the testing done on the algorithms proposed in other papers (the REALGO paper in particular); it is not sufficient to compare a new evolutionary algorithm against others of the same type. It must also be compared against top-performing non-evolutionary algorithms to determine whether overall performance is truly superior.

Another aspect this paper shows is performance. In general, the evolutionary algorithms were unacceptably slow, which can cause even a well-performing algorithm to be useless in practice. This is a point which the more positive papers do not address; for example, the REALGO scheme seems as though it may take a significant amount of time to train, and a constantly-evolving malware detector could potentially place a significant burden on a "live" system. Even detectors which do not constantly evolve, such as IMAD, could become execution bottlenecks depending on how they are implemented. For example, IMAD logs all system calls made by every executable; while this may not be a huge issue in most cases, depending on the application and the number of calls, it could cause performance problems. In practice, even if the evolution only occurs during "scanning", this is when the user is most concerned about performance. While such systems appear to be quite promising in early testing, performance should also be evaluated before genetic algorithm solutions are declared to be superior.

6.2 Specific Examples of Genetic Algorithms in Malware Detection

Despite the somewhat mixed results of the analysis as a whole, there are certain key areas where evolutionary malware detection schemes appear to hold particular promise. In general, these focus around the ability for such systems to detect novel threats quickly, whereas more traditional detection schemes require a waiting period before new threats can be detected. Essentially, the power of genetic algorithms is that they can more efficiently take advantage of prior "learned" knowledge inherent to the system. Being able to quickly recognize new malware that is similar but not identical to previous versions requires that prior knowledge be used; this is in fact one of the major shortcomings of common signature based detection, which essentially treats each piece of malware as a totally separate entity with no relation to other malware. Rather than providing analysis based on the similarities and evolutions of "strains" of malware, each signature operates more or less in a vacuum to target a specific piece of malware. Genetic algorithms appear to offer a large boost in this area as they can be used explicitly to "evolve" signatures in a manner similar to the way that malware "evolves" over time. Rather than ignoring the relations between malware, this technique actually makes use of it to generate new anti-malware signatures.

Even if genetic algorithms are not used directly to evolve methods of detection (as in the REALGO paper [1]), they still offer some significant advantages as they work well for handling real-time data which changes over time. This helps even in more traditional optimization problems (such as for optimizing classification parameters as in the IMAD paper [2]) as it gives the malware detector greater flexibility than a static classifier would have. If, for example, a series of viruses were to appear that were functionally similar but had different code structure, traditional malware detection would require separate signatures for all variants. However, with a detector that can learn the behavior patterns of the malware, it is possible for it to learn to recognize all of the new malware with relatively little trouble. Additionally, a malware detector that learns can be more easily used to prevent threats while they are occurring, rather than based on a static analysis of a potential malware executable. This means that even if a piece of malware has never been seen before (meaning even a genetic algorithm based approach cannot have learned to recognize it) appears, if it behaves in a similar manner to other threats it can be detected in real time and ideally terminated before damage is done. A non-learning detector would have great difficulty doing this, as it would once again require an explicit signature to detect possible malicious activity of a process.

6.3 What About Evolving Malware?

Genetic algorithms clearly have some strong points when it comes to detecting malware, but this is not the only way that they can be applied to the malware paradigm. While malware detectors are learning the structure and behaviors of each new type of malware, malware writers are also constantly changing their "products" in order to evade detection. This, too, can be improved by applying genetic algorithms as demonstrated in the malware evolution paper [3]). While this paper did not focus much on generating malware specifically designed to avoid detection (it was more of a case study of whether evolving malware is possible), it would be fairly trivial to tweak the fitness function used in malware evolution to specifically reward results that are highly evasive. In fact, it would not be difficult to train the malware generator based on how well its resultant programs evaded detection by an evolutionary malware detector! Essentially, introducing genetic algorithms to the area of malware can lead to an artificial "evolutionary arms race" where each side is evolving more and more specialized defenses based on changes made by the "opponent".

While the most obvious fitness function to evolve malware based upon is evasiveness, it is by no means the only attribute that can be optimized for. It would also be possible, in theory, to train a malware generator to produce malware that does maximum damage to a system, or that is the most "contagious", or that is best suited for some specific task (such as stealing passwords). In short, evolvable malware could in theory be "grown" for just about any task the writer desires. The biggest benefit (at least to malware writers) is that by automating the development and testing process, it becomes much easier to generate a large quantity of high-quality malware, even without high expertise in the area. There are already many specialized "cracking" tools available; it is not unreasonable to think that malware generators could join them relatively easily. In the end, while genetic algorithms offer an interesting new area of research into malware detection and mitigation, they also open the door for more easily created high-quality malware. Ultimately, neither side may gain much of an advantage, as the theory behind these advances can be applied easily to either side of the problem.

7 Conclusions

Overall, applying genetic algorithms to malware detection appears to be a promising idea, even if it may have some drawbacks. It is clearly not a panacea, as it can have some serious drawbacks particularly in terms of overhead and execution costs. However, at the same time it offers some significant advances in the way in which malware detection is done, and shows a different paradigm for detection (real-time, based more heavily on learned information, as opposed to comparison to static, slow-to-be-released signatures). Even if the advances offered by genetic algorithms themselves are not immediately applicable, the paradigm shift itself may be useful in making even more traditional malware detection methods more useful. In the long term, it does seem that at least some evolutionary aspects would make a big improvement, however, as the benefits offered are significant and the drawbacks do not seem to be insurmountable. At some point, the current model will become entirely unsustainable, and the ideas offered by evolutionary strategies seem to remedy many of the problems with current schemes.

On the other hand, as malware detectors become more intelligent, it is almost guaranteed that the malware will as well. While evolutionary malware detectors are much more able to deal with rapidly changing threats, malware that can evolve to avoid such detectors is also quite likely to become common. When the malware is capable of evolving based on current threats to it (i.e., malware detectors), it will likely become much harder to detect, as well as likely become more damaging. This is not to say that the threats of evolutionary programs outweigh the benefits, but to point out that improvements will not only come on the "good" side.

7.1 Personal Observations

This paper was quite fascinating to me, as I learned a significant amount about genetic algorithms. Prior to this research, I had a general idea of what genetic algorithms were but did not know much specific about them, nor how they can be applied. I had envisioned them more as a stand-alone idea, not something that could be used in conjunction with other methods (for example, to train weights on a more "traditional" classifier). My concept of them was more along the lines of the REALGO paper, where the output of the algorithm was the final product. I also learned a lot about the wide variety of places that genetic algorithms are applied, and how common they seem to be in a wide variety of learning-based tasks. In addition, it was really fascinating to see what the cutting edge research on malware detection is. As mentioned in the introduction, I have a personal interest in more intelligent malware detection, and am thus quite interested in what research is being done in the area. While the ideas presented clearly have a long way to go to completion, the ideas seem promising and I will be interested to see what comes of them in several years.

The most difficult aspect of this paper for me was synthesizing common observations from the four different papers. Each paper is quite specific, and it took some work to determine where they overlapped. However, when combined together, they give an interesting view into the various issues involved in intelligent malware detection, including the application of the same techniques to generating "better" malware. It also was difficult to get at the deeper "meaning" of the papers, to look beyond the simple results presented to evaluate what the results actually mean in a larger setting. For example, while the results of the REALGO paper were quite fascinating, it was also necessary to step back and look at the issue in light of points raised in other papers (such as the general comparison paper) and really think about what they said as a whole.

References

[1] K. S. Edge, G. B. Lamont, and R. A. Raines. A retrovirus inspired algorithm for virus detection & optimization. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 103–110, New York, NY, USA, 2006. ACM.

[2] S. B. Mehdi, A. K. Tanwani, and M. Farooq. Imad: in-execution malware analysis and detection. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1553–1560, New York, NY, USA, 2009. ACM.

[3] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq. Evolvable malware. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1569–1576, New York, NY, USA, 2009. ACM.

[4] M. Z. Shafiq, S. M. Tabish, and M. Farooq. On the appropriateness of evolutionary rule learning algorithms for malware detection. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2609–2616, New York, NY, USA, 2009. ACM.