Computational Intelligence, Volume 20, Number 3, 2004

DETECTING NEW FORMS OF NETWORK INTRUSION USING GENETIC PROGRAMMING

WEI LU AND ISSA TRAORE

Department of Electrical and Computer Engineering, University of Victoria, Victoria, B.C., Canada

How to find and detect novel or unknown network attacks is one of the most important objectives in current intrusion detection systems. In this paper, a rule evolution approach based on Genetic Programming (GP) for detecting novel attacks on networks is presented and four genetic operators, namely reproduction, mutation, crossover, and dropping condition operators, are used to evolve new rules. New rules are used to detect novel or known network attacks. A training and testing dataset proposed by DARPA is used to evolve and evaluate these new rules. The proof of concept implementation shows that a rule generated by GP has a low false positive rate (FPR), a low false negative rate and a high rate of detecting unknown attacks. Moreover, the rule base composed of new rules has high detection rate with low FPR. An alternative to the DARPA evaluation approach is also investigated.

Key words: genetic programming, network security, intrusion detection, anomaly detection, rule evolution, rule coverage.

1. INTRODUCTION

Intrusion detection has been extensively studied since the seminal report written by Anderson (1980). Traditionally, intrusion detection techniques are divided into misuse detection and anomaly detection. Misuse detection techniques mainly focus on developing models of known attacks, which can be described by specific patterns or sequences of events and data. Anomaly detection techniques model system or users' normal behaviors, and any deviation from the normal behaviors is considered as an intrusion. Misuse detection techniques have low false detection rates (FDR), but their major weakness is that novel or unknown attacks will go unnoticed until corresponding signatures are added to the database of the Intrusion Detection System (IDS). Anomaly detection techniques have the potential to detect novel attacks, but quite often they tend to have high FDR because it is very difficult to discriminate between abnormal and intrusive behavior.

In this paper, we propose a rule evolution approach based on Genetic Programming (GP) (Koza 1992; Wong and Leung 2000) for detecting known or novel attacks on the network. GP extends the fundamental idea of Genetic Algorithm (GA), and evolves more complex data structures. To do so, it uses parse trees to represent initial populations, instead of chromosomes. Moreover, the GP technique can be used to evolve a population of individuals, whereas GA searches the best solution in all possible solutions. Initial rules are selected based on background knowledge from known attacks and can be represented as parse trees. GP will evolve these initial rules to generate new rules. New rules are used to detect novel or known attacks. To evolve and evaluate these new rules, we use the training and testing dataset proposed by DARPA (Lippmann 2000), which includes almost all known network-based attacks, namely *land, synflood, ping of death* (*pod*)*, smurf, teardrop, back, neptune, ipsweep, portsweep,* and *UDPstorm* attacks. The proof of concept implementation shows the GP-based approach can detect smurf and UDPstorm attacks, which are absent from the training dataset. The average false negative rate (FNR) for each rule is 5.04% and the average false positive rate (FPR) is 5.23%. The average rate of detecting unknown attacks for each rule is 57.14%. Moreover, we plot a receiving operator characteristic (ROC) curve of FPR and detection rate when we apply the testing dataset to evaluate our rule base. The ROC

Address correspondence to Wei Lu at the Department of Electrical and Computer Engineering, University of Victoria, P.O. Box 3055 STN CSC, Victoria, B.C., Canada, V8W 3P6; e-mail: wlu@ece.uvic.ca

curve shows that the detection rate will be close to 100% when the (FPR) falls in the range between 1.4% and 1.8%.

The rest of the paper is organized into the following modules. Section 2 presents an overview of related works. Section 3 provides background information on genetic programming. Section 4 discusses how to use GP to generate new rules for detecting known or novel attacks on network. Section 5 presents the evaluation of the new rules using DARPA testing dataset and discusses the experimental results. Section 6 highlights the shortcomings of the DARPA evaluation approach, and then proposes an alternative evaluation approach. Finally, Section 7 makes some concluding remarks.

2. RELATED WORKS

Frank (1994) described and categorized several Artificial Intelligence (AI) techniques that can be used for intrusion detection; use of AI techniques for intrusion detection is categorized according to two dimensions: behavior classification and data reduction. Behavior classification assumes that intrusion can be decided by a given set of known behaviors, and data reduction is typically used to analyze the large amount of audit-log data produced, so as to reduce the amount of data handled by human experts. However, explicit knowledge of known behaviors is difficult to establish. Any mistake occurring in the process of defining patterns of known behaviors will increase false alarm rate and decrease the effectiveness of intrusion detection.

Some early applications of neural networks for user behavior modeling were proposed by Fox et al. (1990). Ghosh, Wanken, and Charron (1998) later extended their idea by using back propagation algorithm for anomaly detection. They established that randomly generating anomalous input data increases the performance of anomaly detection. The biggest limitation of this method is the difficulty of choosing the input parameters. Any mistake in input data selection will increase the false alarm rate. Further, how to initialize the weights of the neural network is still an open question.

Me (1992) initially proposed another application of AI to intrusion detection by using GA for misuse detection. He defined a n-dimensional hypothesis vector H, where $H_i = 1$ if attack i was taking place according to the hypothesis, otherwise $H_i = 0$. Thus, the aim of intrusion detection was reduced to the problem of finding the H vector that maximizes the product $W \times H$, subject to the constraint $AE \cdot H_i <= O_i$. W refers to the n-dimensional weight vector; AE refers to an attacks-events matrix; O refers to the observed n-dimensional audit trail vector. He showed that GA applied to misuse detection has a low false alarm rate. However, this approach cannot identify attacks precisely.

Chittur (2002) extended this idea by using GA for anomaly detection. Random numbers were generated using GA. A threshold value was established and any certainty value exceeding this threshold value was classified as a malicious attack. The experimental result showed that GA successfully generated an accurate empirical behavior model from training data. The biggest limitation of this approach was the difficulty of establishing the threshold value, possibly leading to a high false alarm rate when used to detect novel or unknown attacks.

More works on using GA for intrusion detection are described in Bridges and Vaughn (2000), Balajinath and Raghavan (2001), Gomez et al. (2002). Gomez et al. (2002) proposed a linear representation scheme for evolving fuzzy rules using the concept of complete binary tree structures. GA is used to generate genetic operators for producing useful and minimal structure modification to the fuzzy expression tree represented by chromosomes. This approach, however, required-time consuming training. Bridges and Vaughn (2000) employed GA to tune the fuzzy membership functions and select an appropriate set of features in their prototype IIDS (Intelligent IDS). Balajinath and Raghavan (2001) used GA to learn

individual user behavior. Active user behavior is predicted by GA based on past observed user behavior and used to detect intrusion. In both approaches, the training process is time consuming and they can only be used to detect anomalous behaviors at the host level.

Crosbie and Spafford (1995) employed GP and agent technology to detect anomalous behaviors in a system. The autonomous agents are used to detect intrusions using log data of network connections. Each autonomous agent is used to monitor a particular network parameter and autonomous agents that are predicting correctly are given higher weight value in deciding whether a session is intrusive or not. There are a number of advantages to having many small agents, instead of a single large one. However, communication among these agents is still an issue. Moreover, the training process may be time consuming if the proper primitive for each agent is not chosen.

3. OVERVIEW OF GENETIC PROGRAMMING

3.1. GP Algorithm

GP is an extension of GA (Koza 1992). It is a general search method that uses analogies from natural selection and evolution. The main difference between them is the solution encoding method. GA encodes potential solutions for a specific problem as a simple population of fixed-length binary strings named chromosomes and then applies reproduction and recombination operators to these chromosomes to create new chromosomes. In contrast to GA, GP encodes multipotential solutions for specific problems as a population of programs or functions. The programs can be represented as parse trees. Usually, parse trees are composed of internal nodes and leaf nodes. Internal nodes are called primitive functions, and leaf nodes are called terminals. The terminals can be viewed as the inputs to the specific problem. They might include the independent variables and the set of constants. The primitive functions are combined with the terminals or simpler function calls to form more complex function calls. For instance, GP can be used to evolve new rules from general ones. The rules are represented as *if condition 1 and condition 2 . . . and condition N then consequence*. In this case, the primitive function corresponds to *AND* operator and the terminals are the conditions (e.g., *condition 1, condition 2, . . . , condition N*).

GP randomly generates an initial population of solutions. Then, the initial population is manipulated using various genetic operators to produce new populations. These operators include reproduction, crossover, mutation, dropping condition, etc. The whole process of evolving from one population to the next population is called a generation. A high-level description of GP algorithm can be divided into a number of sequential steps:

1. Create a random population of programs, or rules, using the symbolic expressions provided as the initial population.
2. Evaluate each program or rule by assigning a fitness value according to a predefined fitness function that can measure the capability of the rule or program to solve the problem.
3. Use reproduction operator to copy existing programs into the new generation.
4. Generate the new population with crossover, mutation, or other operators from a randomly chosen set of parents.
5. Repeat steps 2 onwards for the new population until a predefined termination criterion has been satisfied, or a fixed number of generations have been completed.
6. The solution to the problem is the genetic program with the best fitness within all the generations.

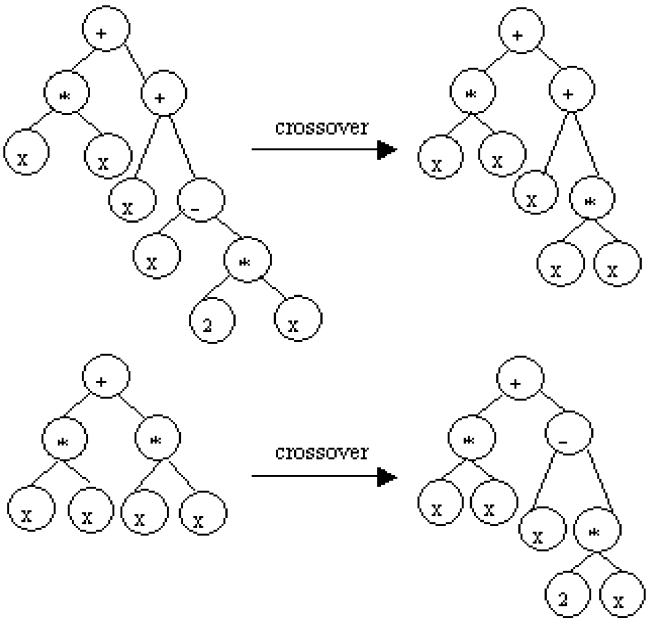

FIGURE 1. Example of crossover in GP.

3.2. Genetic Operators

In GP, crossover operation is achieved firstly by reproduction of two parent trees; two crossover points are then randomly selected in the two offspring trees. Exchanging sub-trees, which are selected according to the crossover point in the parent trees, generates the final offspring trees. The obtained offspring trees are usually different from their parents in size and shape. Figure 1 describes a crossover operation between function $x^2 + x + x - 2x$ and function $2x^2$, they produce two offspring functions $2x^2 + x$ and $x^2 - x$.

Mutation operation is also considered in GP. A single parental tree is firstly reproduced. Then a mutation point is randomly selected from the reproduction, which can be either a leaf node or a sub-tree. Finally, the leaf node or the sub-tree is replaced by a new leaf node or sub-tree generated randomly. Figure 2 describes a mutation operation on function $2x^2$, the produced mutation offspring function is $x^2 + 2x$.

A new operator named "dropping condition" is proposed to evolve new rules in this paper. It randomly selects one condition in the rule, and then turns it into *any*. That is, this particular condition is no longer considered in the rule. For example, the rule

if condition 1 and condition 2 and condition 3 then consequence

can be changed to

if condition 1 and condition 2 and any then consequence

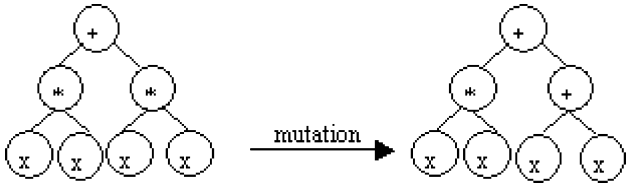

FIGURE 2. Example of mutation in GP.

3.3. Fitness Function

Fitness functions ensure that the evolution is toward optimization by calculating the fitness value for each individual in the population. The fitness value evaluates the performance of each individual in the population. We use a fitness function defined in Wong and Leung (2000) that is based on the support-confidence framework. Support is a ratio of the number of records covered by the rules to the total number of records. Confidence factor (cf) represents the accuracy of rules, which is the confidence of the consequent to be true under the conditions. It is the ratio of the number of records matching both the consequent and the conditions to the number of records matching only the conditions. If a rule is represented as, *if A then B*, and the size of the training dataset is N, then

$$cf = |A \text{ and } B|/|A| \text{ ; } support = |A \text{ and } B|/N.$$

$|A|$ stands for the number of records that only satisfy condition A. $|B|$ stands for the number of records that only satisfy consequent B. $|A \text{ and } B|$ stands for the number of records that satisfy both condition A and consequent B.

A rule with a high confidence factor does not necessarily behave significantly different from the average. Thus, normalized confidence factor is defined to consider the average probability of consequent denoted *prob*.

$$normalized\_cf = cf \times log\ (cf/prob),\ prob = |B\ |/N.$$

To avoid wasting time to evolve those rules with a low support value, a strategy is defined: if support is below a user-defined minimum threshold (min_support), the confidence factor of the rule should not be considered.

Thus, the fitness function is defined as follows:

$$raw\_fitness = \begin{cases} support & if\ support < min\_support \\ w_1 \times support + w_2 \times normalized\_cf & otherwise \end{cases}$$

Where the weights w_1 and w_2 are user-defined and used to control the balance between the confidence and the support during the searches.

Token competition is used to increase the diversity of solutions (Leung and Yam 1992). The idea is as follows: In the natural environment, once an individual finds a good place to live, then (s)he will try to protect this environment and prevent newcomers from using it, unless the newcomers are stronger than this individual. Other weaker individuals are hence forced to search for their own place. In this way, the diversity of the population is increased. A token is allocated to each record in the training dataset. If a rule matches a record, its token will be seized by the rule. The priority of receiving the token is determined by the strength of the rules. Thus, a rule with high raw_fitness score can acquire as many tokens as possible. The modified fitness is defined as follows:

$$modified\_fitness = raw\_fitness \times count/ideal,$$

where count is the number of tokens that the rule has actually seized, ideal is the total number of tokens that it can seize, which is equal to the number of records that the rule matches.

4. GENERATING NEW RULES USING GP

The use of GP to detect unknown attacks is based on the belief that new rules will have better performance than initial ones based on known attacks. Better performance means the

new rules obtained after evolving the initial ones using GP will not only cover known attacks, but also possibly detect the novel ones.

Individual solution in a population is represented as a derivation tree that we describe using a string data structure. For example, a tree can be represented as "*AabAcdAceI*". *A* means "and" operator; a, b, c, d, and e correspond to the conditions in the rules. I is the consequence, which means intrusion. The redundant conditions in the rule will be deleted after the evolution, and thus, *AabAcdAceI* can be interpreted as "*if a and b*, *and*, *c and d and e*, *then intrusion.*" The attribute values of *a,b,c,d,e* are selected from known attacks.

New rules are generated in two phases. In the first step, temporary new rules are composed of new rules generated by four operators including mutation, reproduction, crossover, and dropping condition and additional rules directly generated from previous populations. Thus, the number of temporary new rules is doubled. In the second phase, one half of the temporary new rules with the highest fitness scores after token competition are retained and passed to the next generation.

To assess the feasibility and efficiency of GP for intrusion detection, we have selected an initial population of 40 rules that cover a series of network-based attacks. Table 1 shows 10 instances of the initial rules; the rest of the rules are given in Appendix A.

We calculate the fitness value for each rule based on the training dataset. Currently, the most widely used training and testing dataset for anomaly detection is provided by DARPA Intrusion Detection Evaluation Program (Lippmann 2000), consisting of the raw TCP dump data of 9 weeks activity in a local area network simulating a typical U.S. Air Force LAN. The training dataset is labeled as either normal or intrusive. The test dataset is similar with the training dataset. The only difference is the test dataset includes some unknown attacks not occurring in the training dataset.

In our case, 10,000 network connection records provided by DARPA training dataset are used to train the rules, each connection lasting 2s. Eleven parameters defined in DARPA dataset are used to describe the attacks in the training dataset. Table 2 describes these parameters and their meaning.

The rules in the initial population are evolved using mutation, crossover, and dropping condition operators. The rates of crossover, mutation, and dropping condition operations are respectively 0.6, 0.01, and 0.001 for each rule. Forty offspring rules are evolved from the previous forty parent rules. Based on token competition, combining offspring rules with parent rules generates temporary new rules. One half of the temporary new rules with highest fitness scores after token competition are selected as the new rules.

TABLE 1. Initial Rules

Rules	Meaning
Afp	if land = 1 and wrong_fragment = 0 then intrusion
AAgg	if wrong_fragment > 1 then intrusion
Aab	if protocol_type = tcp and count > 3 then intrusion
bAbA	if srv_count > 3 then intrusion
AhAg	if protocol_type = icmp and wrong_fragment > 1 then intrusion
rA	if synflood = 1.00 then intrusion
AatA	if protocol_type = tcp and num_compromised > 1 then intrusion
Aaav	if protocol_type = tcp and same_srv_rate = 1.00 then intrusion
wwwwA	if diff_srv_rate > 0.33 then intrusion
Aajt	if count<3 and num_compromised > 1 then intrusion

TABLE 2. Representation of Parameters

Parameters	Meaning
protocol_type	Type of protocol
land	Flag to identify whether connection is from/to the same host/port
wrong_fragment	Number of wrong fragments in the connection
synflood	Connections that have "SYN" errors
num_compromised	Number of compromised conditions
same_srv_rate	Percentage of connections to the same services
diff_srv_rate	Percentage of connections to the different services
count	Number of connections from the same source host to the same destination host
srv_count	Number of connections from the same source service to the same destination service
dst_host_count	Number of connections from the same destination host to the same source host
dst_host_srv_count	Number of connections from the same destination service to the same source service

TABLE 3. New Rules

Rules	Meaning
Ag	if wrong_fragment $>$ 1 then intrusion
Afpq	if land $=$ 1 and wrong_fragment $=$ 0 and synflood $=$ 0 then intrusion
Aav	if protocol_type $=$ tcp and same_srv_rate $=$ 1.00 then intrusion
Ahcq	if protocol_type $=$ icmp and dst_host_srv_count $>$ 160 and synflood $=$ 0 then intrusion
Aikq	if protocol_type $=$ udp and srv_count $>$ 367 and synflood $=$ 0 then intrusion
Aat	if protocol_type $=$ tcp and num_compromised $>$ 1 then intrusion
At	if num_compromised $>$ 1 then intrusion
Ag	if wrong_fragment $>$ 1 then intrusion
Ajlpr	if count $<$ 412 and dst_host_count $<$ 810 and wrong_fragment $=$ 0 and synflood $>$ 1 then intrusion
Ail	if protocol_type $=$ udp and dst_host_count $>$ 203 then intrusion

The evolution will not be terminated until we have executed 5,000 runs or the fitness value for each rule is bigger than a threshold equal to 0.95. Table 3 describes 10 instances of obtained new rules. To view the rest of the new rules, please refer to Appendix A.

The initial and new rules are composed of attribute descriptors. Table 4 shows attribute descriptors representations and meanings.

5. EVALUATION OF NEW RULES

Evaluation of intrusion detection approaches for detecting novel attacks is an important and multi-faceted problem. The training dataset we use is one day's connection records provided by DARPA, consisting of 10,000 connection records. Eight kinds of network attacks are included in the training dataset, namely *land*, *synflood*, *pod*, *teardrop*, *back*, *neptune*, *ipsweep*, and *portsweep*. The testing dataset we use is another one-day activity consisting of

TABLE 4. Representation of Terminals

Terminal	Meaning	Terminal	Meaning
s	num_compromised = 0	i	protocol_type = udp
b	count > R1	f	land = 1
c	srv_count > R1	o	land = 0
d	dst_host_count > R1	j	count < R2
k	srv_count < R2	q	synflood = 0
p	wrong_fragment = 0	r	synflood > 1
e	dst_host_srv_count > R1	a	protocol_type = tcp
l	dst_host_count < R2	h	protocol_type = icmp
m	dst_host_srv_count < R2	u	same_srv_rate = 0.00
t	num_compromised > 1	v	same_srv_rate = 1.00
g	wrong_fragment > 1	w	diff_srv_rate > R3

Note: R1, R2, and R3 are random values.

10,000 connection records. Ten kinds of network attacks are included in the testing dataset, namely *smurf*, *UDPstorm*, *land*, *synflood*, *pod*, *teardrop*, *back*, *neptune*, *ipsweep*, and *portsweep*. *Smurf* and *UDPstorm* attacks are the novel attacks which are absent from the training dataset. Detection of attacks involved in the test dataset, and not occurring in the training dataset, assesses the potential ability to detect novel attacks. We use three performance metrics to evaluate the new rules, namely FPR, false negative rate (FNR), and unknown attack detection rate (UADR). A false positive occurs when a rule classifies normal traffic as intrusive. A false negative occurs when a rule characterizes an intrusion as normal. UADR measures the capability of a new rule to detect novel attacks.

For each rule, we calculate its FPR, FNR, and UADR independently. We find that every time we use GP to evolve the rules, the number of generated rules is different and thus the FPR, FNR, and UADR for each rule is also different. Therefore, to statistically evaluate the efficiency of our GP-based approach, FPR, FNR, and UADR are defined as the arithmetical average of the sum of all new rules' rates

$$FPR = average\ (\sum(FPR)_{rules});$$
$$FNR = average\ (\sum(FNR)_{rules});$$
$$UADR = average\ (\sum(UADR)_{rules});$$

For instance, consider a rule base that includes two new rules: rule1 and rule2. The FPR, FNR, and UADR of rule1 is 0.001, 0.015, and 0.56, respectively. The FPR, FNR, and UADR of rule2 is 0.002, 0.03, and 0.78, respectively. Thus, according to the definition:

$$average\ FPR\ for\ each\ rule = (0.001 + 0.002)/2 = 0.0015;$$
$$average\ FNR\ for\ each\ rule = (0.015 + 0.03)/2 = 0.0225;$$
$$average\ UADR\ for\ each\ rule = (0.56 + 0.78)/2 = 0.67;$$

Since the number of new rules is different in each run, the average FPR, FNR, and UADR for each run is also different. We execute 10,000 runs and plot the probability distribution of FPR, FNR, and UADR. Figure 3a illustrates the FPR's probability distribution and Figure 3b illustrates the log scale probability distribution. Figure 4a illustrates the FNR's probability distribution and Figure 4b illustrates the log scale probability distribution. Figure 5a illustrates the UADR's probability distribution and Figure 5b illustrates the log scale probability distribution.

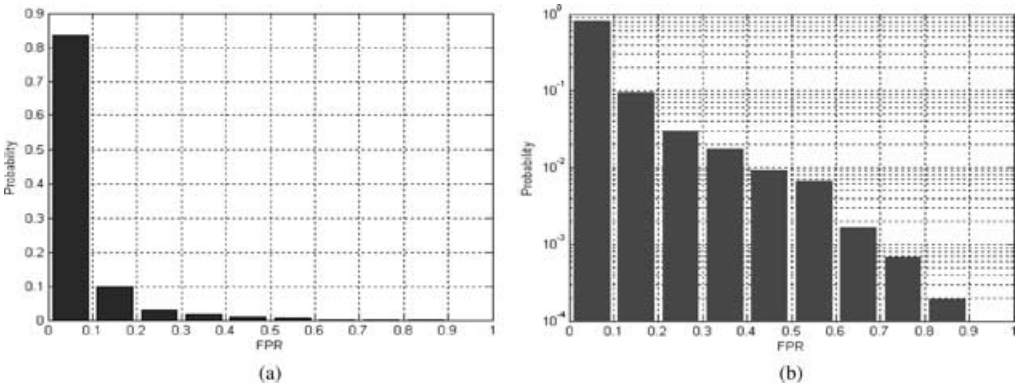

FIGURE 3. (a) Distribution of FPR and (b) log scale distribution of FPR.

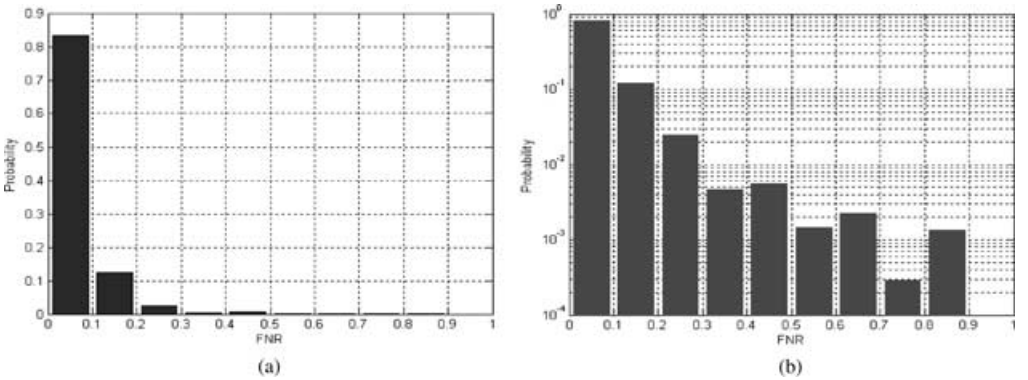

FIGURE 4. (a) Distribution of FNR and (b) log scale distribution of FNR.

The standard deviation of FPR for each rule over 10,000 runs is 0.0944 and the average value of FPR for each rule over 10,000 runs is 0.0523. The confidence interval or, margin of error is 0.0019. In the figure of Log Scale Distribution of FPR, we amplify the probability difference of different FPR scales, and conclude that the probability of the rules whose FPRs

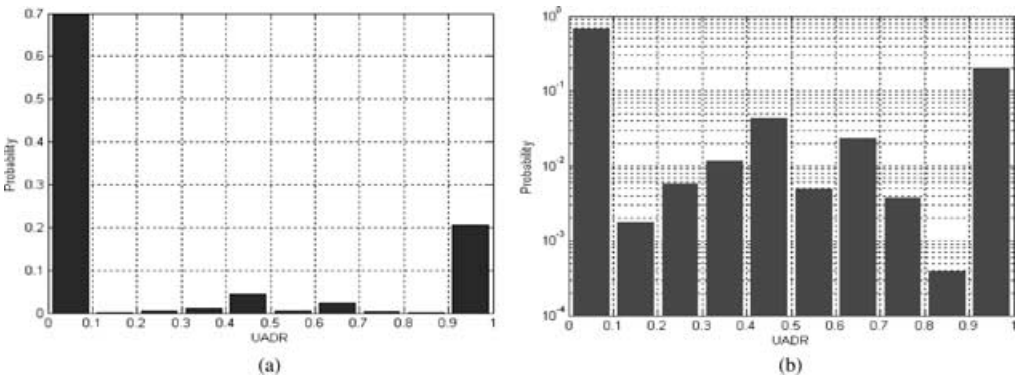

FIGURE 5. (a) Distribution of UADR and (b) log scale distribution of UADR.

fall in a range between 0 and 0.1 is about 10 times greater than the probability of rules whose FPRs fall in other ranges, such as between 0.1 and 0.2, 0.2 and 0.3, etc. Table 5 summarizes this information.

The standard deviation of FNR for each rule over 10,000 runs is 0.0801 and the average value of FNR for each rule over 10,000 runs is 0.0504. The margin of error is 0.0016. In the figure of Log Scale Distribution of FNR, we amplify the probability difference of different FNR scales, and conclude that the probability of the rules whose FNRs fall in a range between 0 and 0.1 is about 10 times greater than the probability of rules whose FNRs fall in other ranges, such as between 0.1 and 0.2, 0.2 and 0.3, etc. Table 6 summarizes this information.

The standard deviation of UADR for each rule over 10,000 runs is 0.397 and the average value of UADR for each rule over 10,000 runs is 0.2509. The margin of error is 0.00794. In the figure of Log Scale Distribution of UADR, we amplify the probability difference of different UADR scales, and conclude that the UADR for each rule in about 20% of the runs are bigger than 0.9 and rates in 70% of the runs fall in a range between 0 and 0.1. Table 7 summarizes this information.

In practical evaluation, we usually use the rule base instead of single rule to test the performance of intrusion detection system based on GP. We execute 10,000 runs to evaluate the statistical performance of our system, since we get different rules every time. We use as evaluation metrics the FPR and the detection rate (DR). Generally speaking, we say that an intrusion detection approach is good if it has high detection rate with low. The probability distribution of FPR for the rule base over 10,000 runs is illustrated in Figure 6a. Figure 6b illustrates the same information for FPR between 0 and 0.1.

The average value of FPR over 10,000 runs is 0.41% and the standard deviation value of FPR over 10,000 runs is 0.0063.

The probability distribution of DR for the rule-base in 10,000 runs is illustrated in Figure 7a. Figure 7b illustrates the log scale probability distribution of DR.

The average value of DR over 10,000 runs is 0.5714, and the standard deviation value of DR over 10,000 runs is 0.4068. Figure 8 is the ROC curve plotting the and FPR the DR.

The DR increases as the FPR does the same. The DR is close to 100% when the FPR is in the range between 1.4% and 1.8%. However, when the FPR is close to 0%, the DR is only about 40%. The DR falls in a broad range from 40% to 100% because the number of rules in the rule base is different for each run. When there are more rules in the rule base, we have a high DR and thus, will possibly have a low FPR. There are some other approaches used to detect intrusion using the DARPA's dataset as a testbed. For example, the ROC curve plotted by Eskin et al. (2000) is illustrated in Figure 9.

Figure 9 shows that DR increases as the FPR does the same. For instance, in the first week the DR is close to 100% when the FPR falls in the range between 0.06% and 0.1%. Eskin's result is better than ours considering the ROC curve comparison. However, the curve plotted by Eskin et al. is for only one kind of attack, a *ftpd attack*, while our ROC curve is for 10 attacks. Our approach can detect 10 kinds of attacks when the FPR is smaller than 1.8%. The approach proposed by Eskin et al. can be used to detect only one kind of attack when its FPR is smaller than 0.1%.

6. ALTERNATIVE EVALUATION METHOD

6.1. Approach

The conventional approach presented in the previous section for evaluating the potential of new rules to detect new forms of attacks consists of some attacks in the testing dataset that are absent from the training dataset, and then testing whether the new rules can detect them or not.

TABLE 5. Scale Distribution Over 10,000 Runs

FPR	0–0.1	0.1–0.2	0.2–0.3	0.3–0.4	0.4–0.5	0.5–0.6	0.6–0.7	0.7–0.8	0.8–0.9	0.9–1.0
Number of run	8,200	1,000	400	200	99	70	20	8	3	0

TABLE 6. Scale Distribution Over 10,000 Runs

FNR	0–0.1	0.1–0.2	0.2–0.3	0.3–0.4	0.4–0.5	0.5–0.6	0.6–0.7	0.7–0.8	0.8–0.9	0.9–1.0
Number of run	8,300	1,100	300	100	100	30	40	5	25	0

TABLE 7. Scale Distribution Over 10,000 Runs

UADR	0–0.1	0.1–0.2	0.2–0.3	0.3–0.4	0.4–0.5	0.5–0.6	0.6–0.7	0.7–0.8	0.8–0.9	0.9–1.0
Number of run	6,835	25	75	105	550	60	250	50	5	2,055

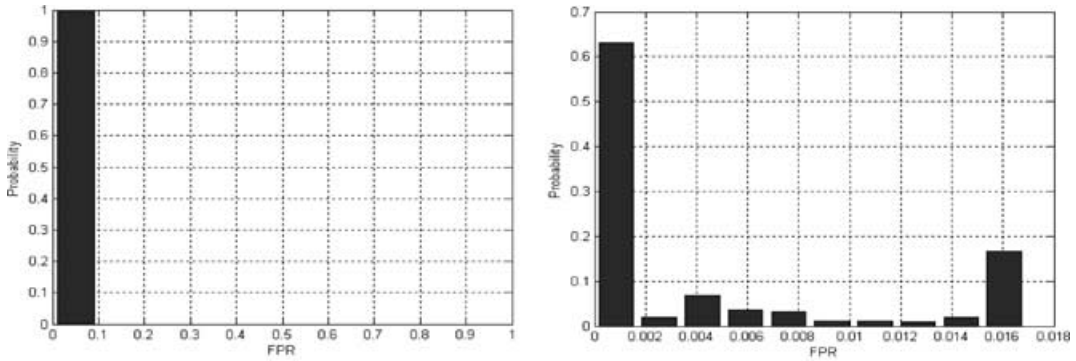

FIGURE 6. (a) Distribution of FPR and (b) distribution of FPR between 0 and 0.1.

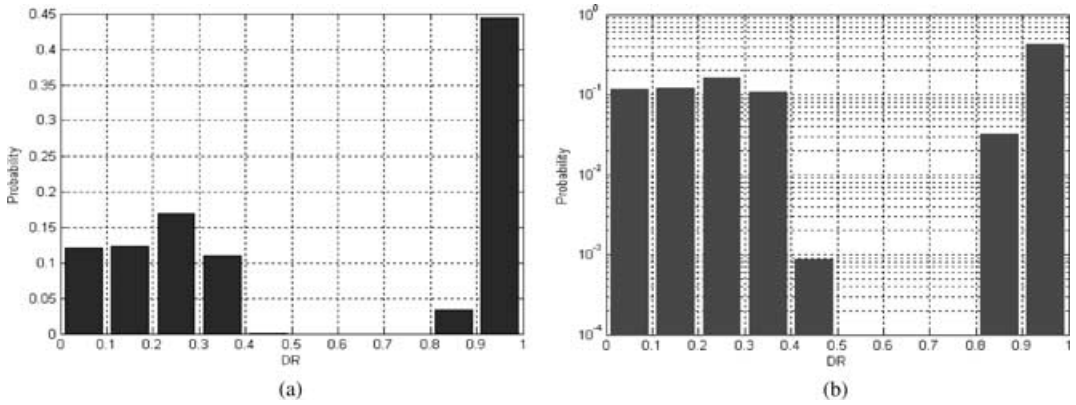

FIGURE 7. (a) Distribution of DR and (b) log scale of distribution of DR.

There are, however, some limitations with this evaluation method. First, it does not consider the completeness of the testing dataset. Completeness refers to the extent to which the dataset represents all types of attacks. For instance, the DARPA's training dataset only contains a total of 24 attack types with an additional 14 types included in the testing dataset.

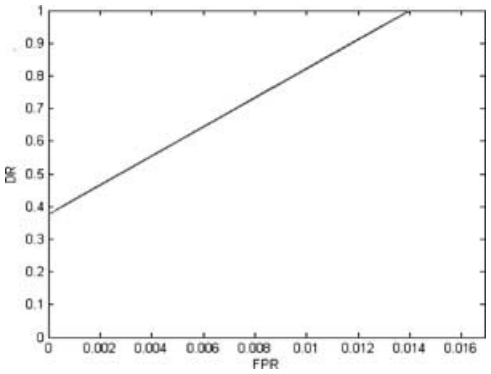

FIGURE 8. ROC curve of FPR and DR.

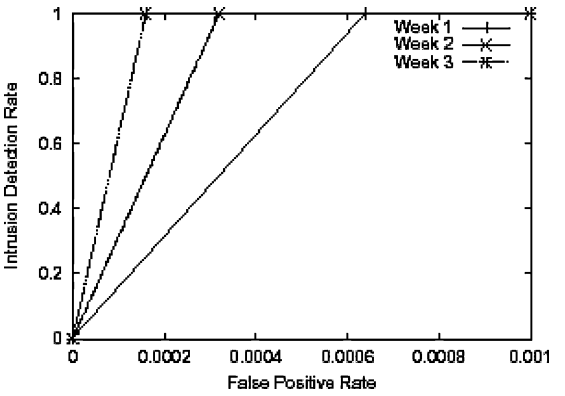

FIGURE 9. ROC curve of FPR and DR plotted by Eskin et al. (2000).

Even if the rules perform well under the testing dataset, they may possibly carry errors that are not identified due to incompleteness of the testing dataset. Second, the coverage of rules by the testing dataset is not taken into account.

An incomplete testing dataset may possibly lead to an inadequate coverage of rules. Consequently, some sections of rules may not be tested due to the absence of test cases. Nevertheless, collecting all types of attacks is very difficult and even if we get all possible kinds of attacks, running them all would be time consuming. A workable alternative consists of improving the completeness of the testing dataset and increasing the coverage of rules.

We propose an alternative evaluation strategy that is depicted by Figure 10. First, we start by building a testing dataset that provides a complete coverage of the new rules generated using GP. We evaluate the coverage of these rules by using a tool named TRUBAC (Testing with Rule-Base Coverage), which implements a coverage analysis method developed by Barr (1995, 1997). Second, we apply the new testing dataset to the initial set of rules from which the new rules have been generated. Test cases that fail the initial rules are flagged as potential new attacks, since they are covered by the new rules but not by the initial ones. Then, we need to verify whether these potential attacks correspond to real attacks. This can be done by reproducing and analyzing corresponding behavior in real network environment.

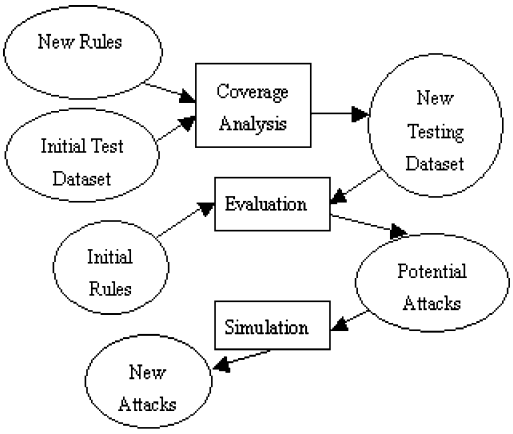

FIGURE 10. Evaluation strategy.

6.2. Coverage Analysis Using TRUBAC

TRUBAC represents a rule base using a directed acyclic graph (DAG). It is based on the premise that a rule-base is also a classification system and exploits the AND/OR graph structure inherent to the rule base. The DAG contains a source node and a sink node, corresponding respectively to the working memory and the classification result. It also contains nodes for *findings* and *classes*. Findings correspond to the antecedents of rules; classes correspond to their consequents. There are also two additional types of interior nodes: subclass nodes and operator nodes. Subclass nodes correspond to intermediate hypotheses; operator nodes correspond to logical operators such as *AND* and *OR*. There are links from the source node to each finding, and from each class to the sink. The antecedent for each rule is represented as a subgraph, which is then connected to the node for the consequent. We refer interested readers to Barr (1995, 1997) for more details. As an example, let us consider the following two rules:

if F1 and F2 and F3 then SC;
if F4 and F5 and SC then C;

The DAG representation of these two related rules is depicted by Figure 11.

TRUBAC defines and implements five rule-base coverage measures (RBCMs), which can be used to guide the selection of testing dataset and give an idea of how well a testing dataset covers some rules. We describe briefly in the sequel these RBCMs.

RBCM1 ensures that there is at least one test data covering one execution path to each class.

RBCM2 ensures that there is at least one test data covering one execution path to each subclass, as well as each class.

RBCM3 ensures that for every finding-class combination, at least one execution path, including this combination, is tested, if such path exists.

RBCM4 provides test data covering at least one execution path that contains a connection from finding to class.

RBCM5 ensures that some test data causes traversal of a collection of execution paths that contains every edge in the graph.

6.3. Generation of New Test Dataset

Based on the five RBCMs, we can analyze the coverage of the new rules and select appropriate test data from the initial test dataset. However, the new rule-base possibly includes some rules uncovered by the initial test dataset. Therefore, some execution paths in the DAG

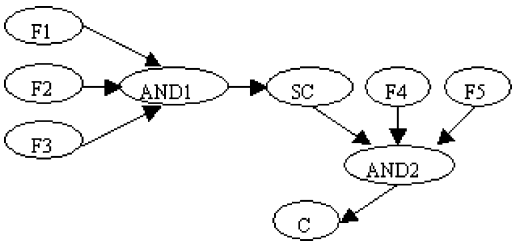

FIGURE 11. Example of DAG representation.

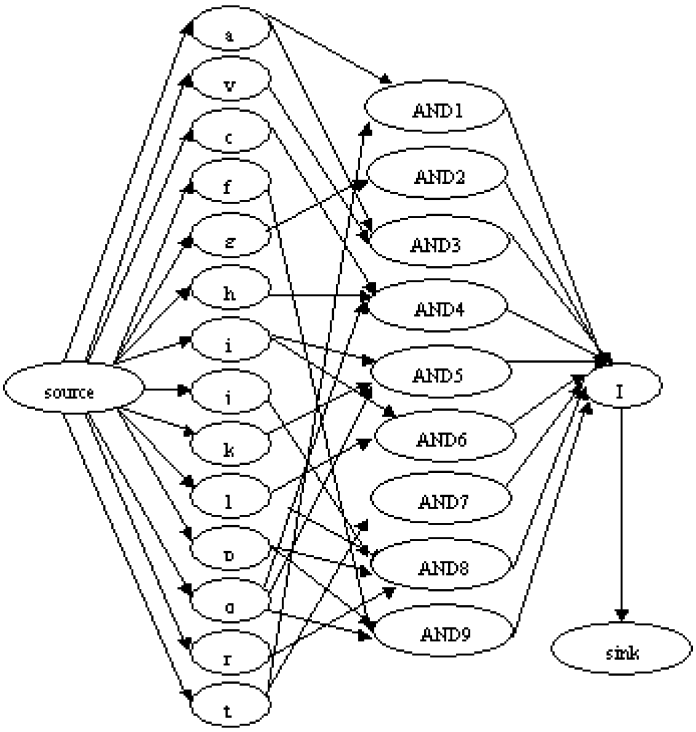

FIGURE 12. Example: DAG of new rules.

may not be covered by any of the available test data. Consequently, we need to update the initial testing dataset by generating additional test data according to the uncovered execution paths. In the sequel, our example illustrates how to construct the new test dataset based on the RBCMs. Consider, for instance, the ten new rules described in Table 3. Since there is no subclass in the rule-base, corresponding DAG can be represented as shown by Figure 12, where a, c, f, g, h, i, j, k, l, p, q, r, and t are findings and I stands for intrusive behavior.

Since there is no subclass, only three RBCMs are applicable in this example: *RBCM1*, *RBCM4*, and *RBCM5*. Based on *RBCM1*, we select one test data from the testing dataset, for instance, a test data covering path "*AND2(g)->I.*" Based on *RCBM4*, we consider two execution paths each including a connection from finding a to class I, "*AND1(a,t)->I*" and "*AND3(a,v)->I.*" From these two paths, we can provide one test data whose path covers the connections from finding a to I. In the same way, we can provide one test data whose paths cover the connections from finding b to I. Ideally, if each path corresponds to one connection, we will have 14 test data, which is a multiple of the number of classes and the number of findings. Table 8 lists all possible execution paths based on each possible connection.

Since the DAG does not include any subclasses, the only difference between *RBCM4* and *RBCM5* is that the test cases provided by *RBCM4* cover only one path for each connection while the test cases provided by *RBCM5* cover all paths for each connection. Finally, we generate 21 test data which can cover all new rules in the rule-base. It is possible that some paths in the DAG may not be covered by any of the available test data. Therefore, to increase the coverage of the new rules, we must synthesize the additional test data according to corresponding path. As an example, if the path "*AND6(i,l)->I*" is not covered by any of the test data, this means "*if protocol_type = udp and dst_host_count > 203 then intrusion.*"

TABLE 8. All Possible Connections and Paths

Connection	Path
finding *a* to class *I*	*AND1(a,t)->I; AND3(a,v)->I*
finding *c* to class *I*	*AND4(h,c,q)->I*
finding *f* to class *I*	*AND9(f,p,q)->I*
finding *g* to class *I*	*AND2(g)->I*
finding *h* to class *I*	*AND4(h,c,q)->I*
finding *i* to class *I*	*AND5(i,k,q)->I; AND6(i,l)->I*
finding *j* to class *I*	*AND8(j,l,p,r)->I*
finding *k* to class *I*	*AND5(i,k,q)->I*
finding *l* to class *I*	*AND6(i,l)->I; AND8(j,l,p,r)->I*
finding *p* to class *I*	*AND8(j,l,p,r)->I; AND9(f,p,q)->I*
finding *q* to class *I*	*AND4(h,c,q)->I; AND5(i,k,q)->I; AND9(f,p,q)->I*
finding *r* to class *I*	*AND8(j,l,p,r)->I*
finding *t* to class *I*	*AND1(a,t)->I; AND7(t)->I*
finding *v* to class *I*	*AND3(a,v)->I*

We simply need to insert manually the packet whose protocol type is UDP into the testing dataset, in order to create corresponding test case.

Overall, 292 test cases are generated for the entire new rule-base involving 40 rules, 276 of which are coming from the initial test dataset and 16 are manually synthesized. Table 9 illustrates the breakdown of new test cases.

6.4. Identifying New Forms of Intrusions

We identify new forms of potential intrusions by evaluating the initial rule-base against the new data-set generated above. Any test case that fails to be covered by the initial rule-base may be considered a potential new attack. From the 292 test cases generated previously, eight failed the initial rules. Consequently, we have eight potential new attacks, which are listed as follows:

'u',0,0,19,20,255,255,0.00,0,0.00,0.00
'u',0,0,20,21,255,255,0.00,0,0.00,0.00
'u',0,0,21,22,255,255,0.00,0,0.00,0.00
'u',0,0,22,23,255,255,0.00,0,0.00,0.00
'u',0,0,23,24,255,255,0.00,0,0.00,0.00
'u',0,0,24,25,255,255,0.00,0,0.00,0.00
'u',0,0,25,26,255,255,0.00,0,0.00,0.00
'u',0,0,26,27,255,255,0.00,0,0.00,0.00

TABLE 9. Breakdown of New Test Cases

	Number of New Test Cases	
Size of Initial Test Dataset	Covered by Initial Rule Base	Uncovered by Initial Rule Base
10,000	276	16

Each test case includes 11 fields, corresponding (as specified in Table 2) respectively to *protocol_type*, *land*, *wrong_fragment*, *count*, *srv_count*, *dst_host_count*, *dst_host_srv_count*, *synflood*, *num_compromised*, *same_srv_rate*, and *diff_srv_rate*.

To verify whether a potential attack corresponds to a real attack, we simulate corresponding behaviors. More specifically, we use a packet generator to simulate corresponding packets in a real network environment, and then analyze the behavior of corresponding hosts, using for instance, a performance analyzer.

6.5. Attack Simulation and Results

It is very difficult to accurately simulate potential attacks on a real network. The eight potential attacks considered above involve continuously sending a defined number of UDP packets from the same source host to the same destination host. Each potential attack lasts 2s. As an example, the first potential attack means that streams of 19 UDP packets are sent from the same source host to the same destination host with a period of 2s. Figure 13 depicts our experimental environment, which consists of the following components:

UDP packet generator: UDPFlood.exe working under Windows NT 4.0.

Source host: Sixteen workstations whose IP addresses range from 142.104.124.101 to 142.104.124.116.

Destination/Victim host: The host provides online computer buying service. Its IP address is 24.77.57.19.

Settings of UDP packet generator: Destination port is set to 8,080; the maximum duration is 0s; packet sending rate is 250 packets/s.

The destination host provides an online computer sale service. The simulation of each of the eight potential attacks results in a denial of service; the website becomes unavailable. The denial of service is created by a flood of UDP packets, which corresponds to an *UDPStorm* attack.

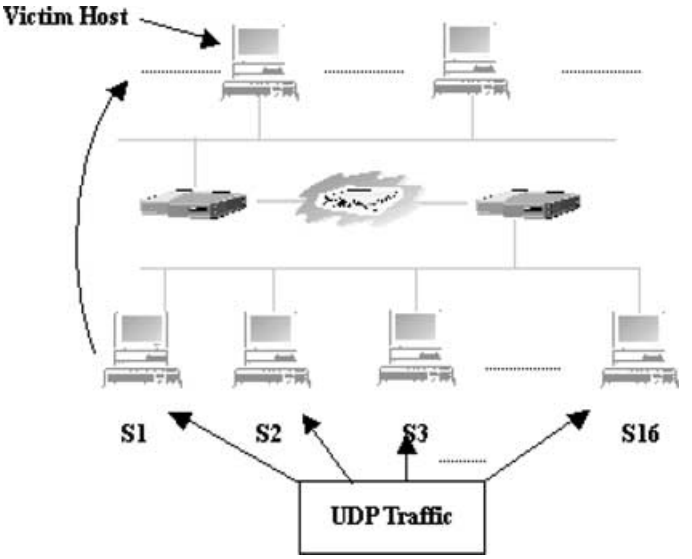

FIGURE 13. Simulated network topology graph.

7. CONCLUSIONS

In this paper, we have presented and evaluated a GP-based approach for detecting known or novel attacks on a network. The proof of concept implementation shows that new rules generated by GP have the potential capability to detect novel forms of attacks. However, the detection result is not good for some runs because the selection of crossover and mutation points in corresponding operations is random. In addition, deciding the probability of genetic operators selection is experience based. In our implementation, the probability of mutation and crossover are 0.01 and 0.6, respectively.

The purpose of the work reported in this paper was mainly to assess the efficiency of GP for known or novel attacks detection. The next step in our work will consist of extending the scope of the rules involved.

REFERENCES

ANDERSON, J. P. 1980, Computer Security Threat Monitoring and Surveillance, Technical Report, James P. Anderson Co., Fort Washington, PA.

BALAJINATH, B., and S. RAGHAVAN. 2001. Intrusion detection through learning behavior model. Computer Communications, **24**(12):1202–1212.

BARR, V. 1995. TRUBAC: A tool for testing expert systems with rule-base coverage measures. *In* Proceedings of the Thirteenth Annual Pacific Northwest Software Quality Conference.

BARR, V. 1997. Rule-based coverage analysis applied to test case selection. Annals of Software Engineering, **4**.

BRIDGES, S. M., and R. M. VAUGHN. 2000. Fuzzy data mining and genetic algorithms applied to intrusion detection. *In* Proceedings of the Twenty-third National Information Systems Security Conference, Baltimore, MD.

CHITTUR, A. 2002. Model generation for an intrusion detection system using genetic algorithm. High School Honors Thesis.

CROSBIE, M., and G. SPAFFORD. 1995. Applying genetic programming to intrusion detection. Technical Report, FS-95-01, AAAI Fall Symposium Series. AAAI Press.

ESKIN, E., and M. MILLER, Z. D. ZHONG, G. YI, W-A. LEE, and S. STOLFO. 2000. Adaptive model generation for intrusion detection systems. *In* Workshop on Intrusion Detection and Prevention, 7th ACM Conference on Computer Security, Athens, GR.

FOX, K. L., R. R. HENNING, J. H. REED, and P. R. SIMONIAN. 1990. A neural network approach towards intrusion detection. *In* Proceedings of 13th National Computer Security Conference, pp. 125–134.

FRANK, J. 1994. Artificial intelligence and intrusion detection: Current and future directions. *In* Proceedings of the 17th National Computer Security Conference, pp. 11–21.

GHOSH, A. K., J. WANKEN, and F. CHARRON. 1998. Detecting anomalous and unknown intrusions against programs. *In* Proceedings of the 14th Annual Computer Security Applications Conference, pp. 259–267.

GOMEZ, J., D. DASGUPTA, O. NASAROUI, and F. GONZALEZ. 2002. Complete expression trees for evolving fuzzy classifiers systems with genetic algorithms and application to network intrusion detection. *In* Proceedings of NAFIPS-FLINT joint Conference, New Orleans, LA, pp. 469–474.

KOZA, J. R. 1992. Genetic Programming. MIT Press.

LEUNG, K. S., and K. F. YAM. 1992. Rule learning in expert systems using genetic algorithms: 1, Concepts. *In* Proceeding of the 2nd International Conference on Fuzzy Logic and Neural Networks, pp. 201–204.

LIPPMANN, R. 2000. The 1999 DARPA off-line intrusion detection evaluation. Computer Networks, **34**(4):579–595.

ME, L. 1992. Genetic algorithms: An alternative tool for security audit trails analysis. Technical Report, Supelec, France.

WONG, M. L., and K. S. LEUNG. 2000. Data mining using grammar based genetic programming and applications. Kluwer Academic Publishers, Netherlands.

APPENDIX

TABLE A1. Initial Rules (Ctd.)

Rules	Meaning
AfpAqA	if land = 1 and wrong_fragment = 0 and synflood = 0.00 then intrusion
AaggqA	if wrong_fragment > 1 and synflood = 0.00 then intrusion
Aigq	if protocol_type = udp and wrong_fragment > 1 and synflood = 0.00 then intrusion
AabrA	if protocol_type = tcp and count > 3 and synflood > 1 then intrusion
Aarc	if srv_count > 3 and synflood > 1 then intrusion
Ahcr	if protocol_type = icmp and srv_count > 3 and synflood > 1 then intrusion
AaaAfj	if protocol_type = tcp and land = 1 and count < 3 then intrusion
AaAoc	if protocol_type = tcp and srv_count > 3 and land = 0 then intrusion
gaAjA	if protocol_type = tcp and count < 3 and wrong_fragment > 1 then intrusion
capAA	if protocol_type = tcp and srv_count > 3 and wrong_fragment = 0 then intrusion
Abc	if count > 3 and srv_count > 3 then intrusion
AatA	if protocol_type = tcp and num_compromised > 1 then intrusion
Akvq	if srv_count < 3 and synflood = 0 and same_srv_rate = 1.00 then intrusion
uagjA	if protocol_type = tcp and same_srv_rate = 0.00 and wrong_fragment > 1 and count < 3 then intrusion
AAvA	if protocol_type = tcp and same_srv_rate = 1.00 then intrusion
AAwv	if protocol_type = tcp and diff_srv_rate > 0.33 and same_srv_rate = 1.00 then intrusion
AqsujaA	if protocol_type = tcp and count < 3 and synflood = 0 and num_compromised = 0 and same_srv_rate = 0.00 then intrusion
Aasf	if num_compromised = 0 and land = 1 then intrusion
AsAg	if wrong_fragment > 1 and num_compromised = 0 then intrusion
AAAtfg	if num_compromised > 1 and land = 1 and wrong_fragment > 1 then intrusion
Aaib	if protocol_type = udp and count > 3 then intrusion
Attt	if num_compromised > 1 then intrusion
Avfrg	if land = 1 and wrong_fragment > 1 and same_srv_rate = 1.00 and synflood > 1 then intrusion
AAAtiA	if protocol_type = udp and num_compromised > 1 then intrusion
AAAvA	if same_srv_rate = 1.00 then intrusion
cwAA	if srv_count > 25 and diff_srv_rate > 0.33 then intrusion
Aaiog	if protocol_type = udp and land = 0 and wrong_fragment > 1 then intrusion
Ar	if synflood > 1 then intrusion
Aagg	if wrong_flagment > 1 then intrusion
AaffA	if land = 1 then intrusion

TABLE A2. New Rules (Ctd.)

Rules	Meaning
Agr	if wrong_fragment > 1 and synflood > 1 then intrusion
Agq	if wrong_fragment > 1 and synflood = 0 then intrusion
Agiq	if protocol_type = udp and wrong_fragment > 1 and synflood = 0 then intrusion
Aw	if diff_srv_rate > 0.33 then intrusion
Aagq	if protocol_type = tcp and wrong_fragment > 1 and synflood = 0 then intrusion
Aqv	if synflood = 0 and same_srv_rate = 1.00 then intrusion
Agu	if wrong_fragment > 1 and same_srv_rate = 0.00 then intrusion
Aadr	if protocol_type = tcp and dst_host_count > 88 and synflood > 1 then intrusion
Ahmq	if protocol_type = icmp and dst_host_srv_count < 160 and synflood = 0 then intrusion
Aicq	if protocol_type = udp and srv_count > 367 and synflood = 0 then intrusion
Aha	if protocol_type = icmp and count > 160 then intrusion
Afs	if land = 1 and num_compromised = 0 then intrusion
Aags	if protocol_type = tcp and wrong_fragment > 1 and num_compromised = 0 then intrusion
Av	if same_srv_rate = 1.00 then intrusion
Af	if land = 1 then intrusion
Afg	if land = 1 and wrong_fragment > 1 then intrusion
Aflp	if land = 1 and sat_host_coun < 203 and wrong_fragment = 0 then intrusion
Afq	if land = 1 and wrong_fragment = 0 then intrusion
Agh	if protocol_type = icmp and wrong_flagment > 1 then intrusion
Abc	if count > 120 and srv_count > 250 then intrusion
Aad	if protocol_type = tcp and dst_host_count > 320 then intrusion
Aijv	if protocol_type = udp and count < 10 and same_srv_rate = 1.00 then intrusion
Akr	if srv_count < 60 and synflood > 1 then intrusion
Aabo	if protocol_type = tp and count > 450 and land = 0 then intrusion
Aeh	if protocol_type = icmp and dst_host_srv_count > 255 then intrusion
Aopt	if land = 0 and wrong_fragment > 1 and num_compromised > 1 then intrusion
Agr	if wrong_fragment > 1 and synflood > 1 then intrusion
Ahfg	if protocol_type = icmp and land = 1 and wrong_fragment > 1 then intrusion
Afw	if land = 1 and diff_srv_rate > 0.5 then intrusion
Act	if srv_count > 46 and num_compromised > 1 then intrusion