IMPROVED OFF-LINE INTRUSION DETECTION USING A GENETIC ALGORITHM

Pedro A. Diaz-Gomez[*]
Ingenieria de Sistemas, Universidad El Bosque
Bogota, Colombia
Email: pdiazg@ou.edu

Dean F. Hougen
Robotics, Evolution, Adaptation and Learning Laboratory (REAL Lab)
School of Computer Science, University of Oklahoma
Norman, OK, USA
Email: hougen@ou.edu

Keywords: Genetic Algorithms, Intrusion Detection, Off-Line Intrusion Detection, Misuse Detection.

Abstract: One of the primary approaches to the increasingly important problem of computer security is the Intrusion Detection System. Various architectures and approaches have been proposed including: Statistical, rule-based approaches; Neural Networks; Immune Systems; Genetic Algorithms; and Genetic Programming. This paper focuses on the development of an off-line Intrusion Detection System to analyze a Sun audit trail file. Off-line intrusion detection can be accomplished by searching audit trail logs of user activities for matches to patterns of events required for known attacks. Because such search is NP-complete, heuristic methods will need to be employed as databases of events and attacks grow. Genetic Algorithms can provide appropriate heuristic search methods. However, balancing the need to detect all possible attacks found in an audit trail with the need to avoid false positives (warnings of attacks that do not exist) is a challenge, given the scalar fitness values required by Genetic Algorithms. This study discusses a fitness function independent of variable parameters to overcome this problem. This fitness function allows the IDS to significantly reduce both its false positive and false negative rate. This paper also describes extending the system to account for the possibility that intrusions are either mutually exclusive or not mutually exclusive.

1 INTRODUCTION

The goal of a security system is to protect the most valuable assets of an organization: data and information. Different organizations will have very different security policies and requirements depending on their missions. This is the case for a bank, an Internet Service Provider, a university, and a consulting firm. However, all have data in some form, and their security mechanisms are tasked with protecting the privacy, integrity, and availability of the data. Many efforts have been made to accomplish this goal: security policies, firewalls, Intrusion Detection Systems (IDSs), anti-virus software, and standards to configure services in operating systems and networks (Bace, 2000). This paper focuses on one of those topics: Intrusion Detection Systems using the audit trail file.

The need for automated audit trail analysis was outlined a quarter century ago (Anderson, 1980) and it is still present. Audit records are used and statistics are gathered and matched against profiles. Information in the matching profiles then determines what rules to apply to update the profiles, check for abnormal activity, and report anomalies detected (Denning, 1986).

A key aspect of an IDS is the hypothesis that exploitation of a system's vulnerabilities is based in the abnormal use of the system (Denning, 1986). In one form or another, all IDSs take into account this assumption. Some, like IDES, NIDES, and EMERALD (Bace, 2000), use statistics, while others use a learning approach like Neural Networks (Bace, 2000), immune Systems (Forrest et al., 1994), Genetic Algorithms (Mé, 1998), and Genetic Programming (Crosbie and Spafford, 1995).

This paper presents a tool to perform misuse detection, using Sun audit trail files (Anonymous, 2000), and uses the guidelines of a previously proposed IDS (Mé, 1998) based on Genetic Algorithms (GAs). To understand this, we first present the basics of computer security (Section 2) and Genetic Algorithms (Section 3). This is followed by a brief introduction to the previous GA–based IDS (Section 4). Next, our own improved IDS is covered thoroughly (Section 5), followed by conclusions and future work (Section 6).

[*]Conducting research at the Robotics, Evolution, Adaptation and Learning Laboratory (REAL Lab), School of Computer Science, University of Oklahoma.

2 COMPUTER SECURITY

An *Intrusion Detection System* (IDS) is a system that monitors and detects intrusions, or abnormal activities, in a computer or computer network. The IDS reports corresponding alarms and may take immediate action on the intrusions (Tjaden, 2004).

An *intrusion* is defined as an attempt to gain access to a system by an unauthorized user. *Misuse* refers to attempts to exploit weak points in the computer or the abuse of existing system privileges. *Abnormal activity* means significant deviations from the normal operation of the system or use of the system by users (Tjaden, 2004).

Intrusion detection, then, is the process of monitoring computer networks and systems for violations of security policy. In the simplest terms, intrusion detection systems consist of three functional components:

1. an information source that provides a stream of event records,
2. an analysis engine that finds signs of intrusions, and
3. a response component that generates reactions based on the outcome of the analysis engine (Bace, 2000).

In order to get information for intrusion analysis, an *audit trail* is often used. According with the Rainbow Series of computer security documents, outlined by the Department of Defense (Bace, 2000), the goals of the audit mechanism are:

- to allow the review of patterns of access,
- to allow the discovery of both insider and outsider attempts to bypass protection mechanisms,
- to allow the discovery of a transaction of a user from a lower to a higher privilege level,
- to serve as a deterrent to users' attempts to bypass system-protection mechanisms, and
- to serve as a yet another form of user assurance that attempts to bypass the protection will be recorded as discovered.

The need for automatic audit trail review to support security goals has been well documented with the matrix in Table 1 suggested for classifying risks and threats to computer systems (Anderson, 1980).

	Not authorized to use data/program	Authorized to use data/program
Not authorized to use computer	CASE A External Penetration	Blank
Authorized to use computer	CASE B Internal Penetration	CASE C Misfeasance

Table 1: Threat matrix. Redrawn with minor modifications from Anderson, 1980.

This suggests a taxonomy for classifying risks and threats to computer systems that differentiates between external and internal sources of problems. This articulation has been useful in structuring requirements for audit trail content (Bace, 2000). According to this classification, this paper focuses on internal penetration—an audit trail file of a authorized user is analyzed in order to get misuse.

A formal definition of *security* says that it must guaranty confidentiality, integrity, and availability. *Confidentiality* refers to the fact that the information is only known by authorized users. *Integrity* means that the information is protected from alteration. *Availability* means that the system operates as it was designed; it means, for example, that users have access to it when they need it, where they need it, and in the form they need it.

Another crucial aspect of any system's security is its security policy. A *security policy* is the set of practices that is explicitly stated by an organization in order to protect sensitive information (Crosbie and Spafford, 1995).

The content of most security policies is driven by a desire to address threats. A *threat* is defined as any event that has the potential to harm a system. This harm can be access of data by an unauthorized user, destruction or modification of data, or denial of service (Bace, 2000).

Security problems in computer systems result from vulnerabilities. *Vulnerabilities* are weaknesses in systems that can be exploited in ways that violate security policy. Although threat and vulnerability are intrinsically related, they are not the same. Threat is the result of exploiting one or more vulnerabilities. Intrusion detection is designed to identify and respond to both (Bace, 2000).

IDSs can be classified as host-based, multihost-based, and network-based (Tjaden, 2004). *Host-based* IDSs monitor a single computer using the audit trail of the operating system whereas *network-based* IDSs monitor computers on a network by scrutinizing the audit trail of multiple hosts and network traffic.

A *multihost-based* IDS analyzes data from multiple computers. Usually a module of the IDS runs on each individual computer and sends reports to a special module, sometimes called a director, running on one machine. Since the director receives information from the other computers, it can correlate this information to recognize intrusions that host-based systems would probably miss, such as worms. A host-based IDS may not notice that type of intrusion. A multihost-based IDS, with its data from a number of different computers, would have a much better chance of recognizing a worm as it spreads (Tjaden, 2004).

This paper deals with a host-based IDS, and an audit trail file generated by a Sun machine is analyzed.

3 GENETIC ALGORITHMS

Genetic Algorithms (GAs) belong to the field of evolutionary computation and have been widely used in problems that require searching through a huge number of possibilities for solutions (Mitchell, 1998). The strength behind GAs resides in the fact that the search space is traversed in parallel by proposing solutions, at the beginning randomly generated, and those solutions are continuously evaluated with a *fitness function*.

Each set of solutions proposed is called a *population*, and the first set is called the *initial population*. Once all members in the initial population are evaluated and assigned a *fitness value*, the *selection* mechanism is applied in order to produce the *next generation*.

The selection mechanism is applied in order to choose parents that are going to reproduce. Often parents are selected proportionally to their fitness value. One common way to conceive of this is to imagine mapping the fitness value of each individual to a *roulette wheel*, where a greater fitness equates to a larger space on the wheel. Once parents are chosen, *crossover* and *mutation* are applied to them in order to produce *offspring*.

Crossover exchanges subparts of parents (typically two), while mutation changes randomly a particular part of an individual. The process of selection, crossover, and mutation gives the next generation, and the process is repeated until a solution is found or until resources are exhausted.

4 *GASSATA*

One motivation for the topic of this research was the paper "*GASSATA*, A Genetic Algorithm as an Alternative Tool for Security Audit Trail Analysis" (Mé, 1998), so let us analyze the theory behind it.

GASSATA is an off-line tool that increases security audit trail analysis efficiency. The goals of this approach are the following:

- to investigate misuse detection, i.e., to determine if the events generated by a user correspond to known attacks, and
- to search in the audit trail file for the occurrence of attacks by using a heuristic method, Genetic Algorithms, because this search is an NP-complete problem.

This approach is shown schematically in Figure 1.

The audit subsystem recognizes various kinds of events (such as changing to a particular directory or copying a file) which are recorded in the audit trail file. The *Syntax Analyzer* classifies those audit events

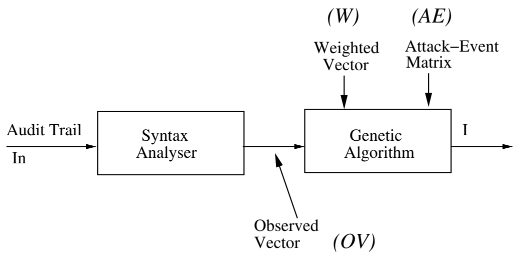

Figure 1: Prototype of *GASSATA*.

and generates the *Observed Vector* (OV). The *Genetic Algorithm* module finds the hypothesized vector I that maximizes the product $W \cdot I$, subject to the constraint $(AE \cdot I)_i \leq OV_i$ profiles where W is a *weight vector* that reflects the priorities of the security manager, AE is the *Attacks-Event* matrix that correlates sets of events with known attack profiles, $1 \leq i \leq N_a$, and N_a is the number of known attack profiles. The duration of the audit session was chosen as 30 minutes and four (4) types of user were defined: inexperienced, novice developer, professional, and UNIX–intensive user.

The fitness function suggested in *GASSATA* is:

$$F(I) = \alpha + \sum_{i=1}^{N_a} W_i * I_i - \beta * T^2. \tag{1}$$

The goal of each component of the equation is as follows:

- α: To maintain $F(I) > 0$, and therefore maintain diversity in the population.
- I: To allow for hypotheses of attacks. The system is rewarded for hypothesizing attacks, particularly those of greatest concern to the security manager. This is generated randomly in the first generation.
- β: To provide a slope for the penalty function.
- *T*: To count the number of times $(AE \cdot I)_i > OV_i$. The system is penalized for hypothesizing sets of attacks that could not have occurred, given the observations.

Experimental results for simulated attacks have been reported (Mé, 1998).

5 AN IMPROVED IDS

As outlined before, intrusion detection systems consist principally of three functional components:

1. an information source that provides a stream of event records,
2. an analysis engine that finds signs of intrusions, and

3. a response component that generates reactions based on the outcome of the analysis engine (Bace, 2000).

In order to get information for intrusion analysis, an *audit trail* is often used. In this research we used the Sun audit trail file (Anonymous, 2000).

5.1 The Security Audit Trail

Security auditing is the formal tracking and analysis of actions taken by computers' users. This process is necessary in order to control current security policies and detect anomalies, abuse, or misuse of the system by *users*.

In order to provide individual user *accountability*, the computing system identifies and authenticates each user. Besides that, computers provide the possibility to register actions taken by users. *Audit data* corresponds, then, to recorded actions taken by identifiable users, associated under the unique user identifier (ID). All processes and activity made by users are recorded in the *audit trail file*. In a sense, audit data is the complete recorded history of a users' system activity that can be gathered for an after-the-fact investigation or to determine the effectiveness of existing security controls (Anonymous, 2000).

The audit trail data used in this research was downloaded from the MIT Lincoln Laboratory (Fried and Zissman, 1998), and it has activity from various users. Since *GASSATA* (Mé, 1998) works its input by user, the file from was filtered by user.

The Solaris audit subsystem, called the SunSHIELD Basic Security Module (BSM), is the module responsible for the accountability of users' activity on the system. The file generated by the BSM has records that describe user-level and kernel events. Each record consists of tokens that identify the process that performed the event, the objects on which it was performed, and the objects' attributes, such as the owners or modes.

5.2 The Analysis Engine

Once the audit data is recorded, it must be reviewed on a regular basis in order to maintain effective operational security. Administrators who review the audit data must watch for events that may signify abnormal use of the system. Some examples include:

- trying to change sensitive information on records of files requiring higher privilege,
- killing critical processes,
- trying to access different user's files,
- probing the system,
- installing of unauthorized, potentially damaging software, and
- exploiting a security vulnerability to gain higher or different privileges.

In order to provide system administrators with the ability to effectively audit user actions, the software developed, as suggested for *GASSATA* (Mé, 1998), provides the capability to read the audit trail and perform analysis of those records based on known intrusions. The program developed to simulate *GASSATA* consists of two parts: the first one, the *scanner*, reads the audit trail to classify and count the user's events, and the second one, the *analysis engine*, is a Genetic Algorithm. In schematic form, the software developed has the same architecture shown in Figure 1.

5.2.1 Data Structures

The scanner looks at all the audit trail records for a user and counts the different kind of events. The output of this program is an array of size N_a called the *Observed Vector* (OV). Each position of this array corresponds to the number of occurrences of an event according with the classification stated before.

The Genetic Algorithm receives as input the Observed Vector and the Attack-Events matrix (*AE*) of known attacks (Mé, 1998). In Table 2, each row corresponds to an event type. Each numeric column—i.e., 0 through 23—corresponds to a known attack; column I corresponds to a hypothesis that is being evaluated, column $AE * I$ corresponds to the multiplication of matrix AE with hypothesis vector I, column OV corresponds to the actual number of events that occurred—this is the output of the scanner program, and column entry i in T is set to 1 if $(AE \cdot I)_i > OV_i$, i.e., if the number of hypothesized events of type i are greater than the actual number of events of that type registered in the audit trail file. Column T' shows the number of of hypothesized attacks that require more events of type i than actually occurred (see Section 5.2.3).

The Observed Vector contains the number of system events performed by a user in a session. As shown in Table 3, the first row is the type of event and the second row is the number of events of that type.

The first position of row 2 shows 1, which means that there was a 1 event of type `User_login_fail`, the eighth position shows 40, which means that there were 40 events of type `ls_fail`. Other event types are listed elsewhere (Mé, 1998).

5.2.2 The Genetic Algorithm

The *analysis engine* is a Genetic Algorithm that works as follows:

```
Generation of the First Population
Do
```

A T T A C K #

E V E N T #

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	I	AE * I	OV	T	T'
0	3																								0	0	0		
1					1																				0	0	0		
2				1																					0	1	0	1	1
3		3																							1	0	0		
4			3																						0	0	0		
5		3							8																1	8	0	1	1
6						5														1		5			1	10	0	1	2
7						30																			1	30	76		
8							5																		1	5	0	1	1
9										3															0	0	0		
10											2														1	2	20		
11												3													1	3	0	1	1
12															10	1									1	0	0		
13																1									0	0	6		
14																1									0	0	0		
15																								4	0	4	4		
16																				1					0	0	0		
17		3				35	5		8	3	2	3			10	3		300		2		5		4	0	62	94		
18																			100						1	100	0	1	1
19								5																	0	5	42		
20														10											1	0	0		
21																				1					1	0	0		
22																									0	0	0		
23																						5			1	5	5		
24																							1			0	0		
25					1																3					3	459		
26													30													30	1335		
27																	50									0	0		

Table 2: The Attack Events matrix *AE*, an example hypothesis Vector I, the resulting Multiplication Vector MV ($= AE * I$), an example Observed Vector OV, and the count of overestimates in columns T and T'.

```
    Selection of Parents
    Reproduction with crossover and mutation
Until an acceptable Solution is found or
    resources are exhausted
```

An *individual (I)* in the population is an array of 24 positions. Each position corresponds to an attack that is being hypothesized.

The *First Population* is generated randomly. The algorithm is guessing the possible occurrence of attacks. 40 individuals are generated.

Selection of Parents. Parents are selected according to their fitness value. The fitness function guides the algorithm to find the possible intrusions.

Crossover and Mutation. Parents that are chosen are crossed with a probability equal to 60% and mutated with a probability of 2.4%. Mitchell (1998) suggests values like 70% for crossover and 0.1% for mutation as commonly appropriate.

Number of generations. The number of generations for each test was variable. At the very beginning of our experiments, for example, we keep track of the fitness values and stop after 500,000 generations; we found, then, the greatest fitness value of all the generations and used it with a threshold equal to STDV$(Max_fitness_value)/4$, so the algorithm begins to run, and if the fitness value goes less than $Max\_fitness\_value)/4 -$ STDV$(Max_fitness_value)/4$ the algorithm stops.

The Attack-Event matrix, which gives the number of events by attack, is multiplied by the individual I that hypothesized the occurrence of attacks. The result is the Multiplication Vector (*MV*) that gives the classified total number of events hypothesized ($= AE * I$). This vector is evaluated using the Observed Vector that records the number of events that have occurred. Each position of the result vector *MV* is compared with the corresponding position in OV. If MV_i, the hypothesized number of events of type i, is less than OV_i (the audit trail observed number of events of type i), then the hypothesis is possible.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	12	6	0	4	9	11	40	34	9	5	45	0	0	2	7	0	29	0	0	0	0	0	0	0	0	0	0

Table 3: An example of an Observed Vector.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4: An example of an Individual. Positions 4, 5, and 8 that have 1 are showing the possible occurrence of attacks 4, 5, and 8.

But if, on the contrary, MV_i is greater than OV_i, that means that the algorithm hypothesized more intrusions than actually occurred, a situation that must be take into account using a penalty factor in the fitness function. The penalty term is then related with the number of events in MV_i that are greater than the observed events in OV_i.

5.2.3 An Improved Fitness Function

A genetic algorithm needs a fitness function that combines objectives and constraints into a single value (Coello, 1998). The problem is not only to find the appropriate function but also to provide accurate values to the parameters that produce the correct solution to the problem for as many instances as possible. It appears that the fitness function proposed in *GASSATA*, a combination of objectives and constraints into a single value using arithmetical operations, should be correct. However, it is difficult to set the parameters so that the algorithm finds intrusions and converges.

For this problem *GASSATA* uses a fitness function that has principally two terms: $\sum_{i=1}^{N_a} W_i * I_i$ that is rewarding, and $\beta * T^2$ that is penalizing.

As the fitness function is giving a payoff to the highest valued individual, the term $\sum_{i=1}^{N_a} W_i * I_i$ is guiding the solution to have the maximum number of intrusions. However, this is good enough until the correct set of intrusions are found. Later on, i.e., if more intrusions than that are hypothesized, the problem of false positives occurs. On the other hand, the term $\beta * T^2$ is diminishing the fitness but in doing so various intrusions can hit the same event. When this happens the counting of overestimates is wrong. These two facts were tested and false positives and false negatives were found. (Results of this testing and an analysis of the reason for it will be published elsewhere (Diaz-Gomez and Hougen, 2005). For the present work, we concentrate on improvements to the IDS).

The solution proposed has two parts:

1. remove the positive term $\sum_{i=1}^{N_a} I_i$, and
2. count overestimates in the correct way; this means, if two intrusions require the same event in numbers in excess of the number of actual events then count them both, and so forth. Call this T'.

With this in mind, the fitness function only has one term, the penalty function and the new fitness function suggested is

$$F(I) = N_e - T' \tag{2}$$

where N_e corresponds to the total number of classified events. For testing this value is 28. T' corresponds to the number of overestimates, i.e., the number of times $(AE \cdot I)_i > OV_i$.

It must be taken into account that the role of α corresponds now to N_e and that β is equal to one. However, the term $\sum_{i=1}^{N_a} I_i$ was suppressed, as stated before. It must be reinforced that the reason for doing that is because the term $\sum_{i=1}^{N_a} I_i$ is giving the number of intrusions hypothesized but those intrusions have not been evaluated yet. In doing the evaluations, that may produce an incorrect count of overestimates.

Now, the hypothesized vector I is really evaluated in T'; the better the hypothesized vector, the smaller T' is, and of course, $F(I) \to N_e$, the maximum. The fitness function is evaluating only the T' term. There is a maximum when $T = 0$.

In this study we divided the set of intrusions into two subsets: *mutually exclusive* and *not mutually exclusive* intrusions. We define mutually exclusive intrusions those that can not occur at the same time, depending of the Observed Vector in the analysis, i.e., if in considering each intrusion alone $T = 0$ but in considering them at the same time $T \neq 0$. That is the case for all intrusions that share some event type; for example, intrusions number 5, 19 and 21, in each of which event number 6 is present (see Table 2).

As our genetic algorithm runs, it creates aggregate solution sets of all possible compatible intrusions found. To do this, it records all the realistic solutions (those where $T = 0$) found in the search space and keeps track of each intrusion it finds within each realistic solution. The algorithm then checks if the intrusion already exists in its current solution set and, if it does not, then it checks if it is mutually exclusive or not in order to add it to the corresponding aggregate solution set. In this way, the algorithm builds up sets of all compatible solutions. This mechanism can be seen as a replacement for the positive term $\sum_{i=1}^{N_a} I_i$ which was misleading the genetic algorithm in *GASSATA*.

The results found with this fitness function are shown in Table 5. The data corresponding to users 2051 and 2506 was extracted by our scanner from audit data files downloaded from the Lincoln Laboratory at MIT (Fried and Zissman, 1998).

	Average Count			Average %		
User	False +	False -	Detected	False +	False -	Detected
2051_7	0	0	3	0	0	100
2051_11	0	0	4	0	0	100
2506_15	0	0	4	0	0	100
Zero Vector	0	0	0	0	0	100
One Intrus.	0	0.1	0.9	0	10	90
Two Intrus.	0	0	2	0	0	100
Three Intrus.	0	0	3	0	0	100

Table 5: Results with fitness function $F(I) = N_e - T$ averaged over 10 runs.

As can be seen, with the fitness function proposed *there are no false positives* and the number of *false negatives decreases dramatically*. This time 70 runs were performed with different data and only one time a false negative was present.

These results are significantly better than any of those found with any of the parameters we tested for the original *GASSATA* fitness function. For example, setting $\alpha = N_e^2/2$, $\sum_{i=1}^{N_a} W_i > N_e^2/2$, and $\beta = 1$ resulted in the performance shown in Table 6.

	Average Count			Average %		
User	False +	False -	Detected	False +	False -	Detected
2051_7	8.9	0.2	2.8	297	7	93
2051_11	9.3	0.0	4.0	233	0	100
2506_15	8.6	0.8	3.2	215	20	80
Zero Vector	10.0	0.0	0.0	Inf.	0	100
Full Vector	0.0	10.9	13.1	0	45	55

Table 6: Results with Fitness Function as in Equation 1 using $\alpha = N_e^2/2$, $\sum_{i=1}^{N_a} W_i > N_e^2/2$, and $\beta = 1$.

5.3 Improvements

As with many heuristic tools, we have difficulties in the implementation of *GASSATA*. Some difficulties arise in providing accurate values for the fitness parameters α, W, and β. In doing that we found a high percentage of false positives, so this study made the following improvements in order to:

- dismiss false positives and false negatives,
- find the maximum set of intrusions and disaggregate them as mutually exclusive or not,
- record all events not considered in the intrusion analysis.

As was explained in Section 5.2.3 the term $\sum_{i=1}^{N_a} W_i * I_i$ proposed for the fitness function in Equation 1 is guiding the algorithm to give false positives, so with the new fitness function proposed in this research (see Equation 2), we have discarded that term. Experimentally, as is shown in Table 5, there were no false positives with our fitness function $F(I) = N_e - T$.

We also managed to all but eliminate false negatives by building up aggregate solution sets of all compatible intrusions found, as also explained in Section 5.2.3.

Another improvement is related with the recording of events not considered in the analysis. That is, the scanner is looking for predefined events; if there are events not defined, those are recorded, so the system administrator can check them and evaluate if there are critical events not considered and that can be included in the scanner.

6 CONCLUSIONS AND FUTURE WORK

This paper proposes a fitness function independent of variable parameters, making the fitness function to solve this particular problem quite general and independent of the audit trail data. This approach can be generalized to similar multi-objective fitness functions for genetic algorithms. At the same time the system proposed improves the one suggested previously (Mé, 1998) by recording all the events not considered in the intrusion analysis, finding the maximum set of intrusions and disaggregating them as mutually exclusive or not, and diminishing the false positives and false negatives.

One topic that can be addressed in future work is to investigate system performance using different crossover and mutation rates.

We also consider it important for the future of intrusion detection systems to consider the standardization of audit trail files. Such a standard has the principal benefit that it would enable logs generated by different operating systems to be reconciled. With the audit trail standardized, the analysis of logs by a central engine would be simpler, because that engine would deal with only one file structure.

Another improvement of the specific intrusion system developed is the use of real-time intrusion detection, because such a system can catch a range of intrusions like viruses, Trojan horses, and masquerading before these attacks have time to do extensive damage to a system.

This paper presented some of this new research in intrusion detection, by using a GA as an analytical engine that performs intrusion detection. However, the field of IDSs is quite diverse and other approaches such as immune systems and neural networks have been developed in order to improve this mechanism.

The field is deep and there are promising new ways to think about it. Evolutionary computation offers a chance to see intrusion detection systems with the ability to evolve—evolution that could sometimes exceed human programmers. There are new paradigms to explore and we can use computers themselves as the vehicle.

REFERENCES

Anderson, J. P. (1980). Computer security threat monitoring and surveillance. Technical Report 79F296400, James P. Anderson, Co., Fort Washington, PA.

Anonymous (2000). SunSHIELD basic security module guide (Solaris 8). Technical Report 806-1789-10, Sun Microsystems, Inc., Palo Alto, CA. http://docs.sun.com/db/doc/806-1789, accessed July 2004.

Bace, R. G. (2000). *Intrusion Detection*. MacMillan Technical Publishing, USA.

Coello, C. A. C. (1998). A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, 1(3):269–308.

Crosbie, M. and Spafford, G. (1995). Applying genetic programming to intrusion detection. In *Papers from the 1995 AAAI Fall Symposium*, pages 1–8.

Denning, D. E. (1986). An intrusion-detection model. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 118–131.

Diaz-Gomez, P. A. and Hougen, D. F. (2005). Analysis of an off-line intrusion detection system: A case study in multi-objective genetic algorithms. In *Proceedings of the Florida Artificial Intelligence Research Society Conference*, to appear. AAAI Press.

Forrest, S., Perelson, A. S., Allen, L., and Cherukuri, R. (1994). Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 202–212, Oakland, CA. IEEE Computer Society Press.

Fried, D. and Zissman, M. (1998). Intrusion detection evaluation. Technical report, Lincoln Laboratory, MIT. http://www.ll.mit.edu/IST/ideval/, accessed March 2004.

Mé, L. (1998). GASSATA, a genetic algorithm as an alternative tool for security audit trail analysis. In *First International Workshop on the Recent Advances in Intrusion Detection*, Belgium.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press.

Tjaden, B. C. (2004). *Fundamentals of Secure Computer Systems*. Franklin and Beedle & Associates.