# Three Approaches to Intrusion Detection. Analysis and Enhancements.

Pedro A. Diaz-Gomez and Dean F. Hougen
Robotics, Evolution, Adaptation, and Learning Laboratory (REAL Lab)
School of Computer Science, University of Oklahoma, OK., USA.
pdiazg@ou.edu hougen@ou.edu

***Abstract*— One of the most important responsibilities of every company is to preserve the integrity, confidentiality, and availability of its data. Many efforts have been made to accomplish this goal: security policies, firewalls, intrusion detection systems, anti-virus software, and standards to configure services in operating systems and networks. This paper focuses on one of those topics: intrusion detection systems. First Denning's classic model for intrusion detection is introduced, then two examples for doing intrusion detection are presented that follow the guidelines of a new approach: the use of evolutionary computation.**

## I. Introduction

This section provides basic definitions related to computer security. It deals with the importance of protecting data from intruders. After that we introduce genetic programming and genetic algorithms. This specific scope is adopted in order to introduce topics of this paper related to evolutionary computation applied to intrusion detection.

### A. Computer Security

Computers are systems that store, process, and retrieve data. Many activities in our modern world deal with computers as tools to improve the use of data. Data is a invaluable resource for every company or enterprise, so it must be protected against intruders or unauthorized users.

Computer security is the science that studies the protection of data. Data in computers can be added, modified, and/or deleted; and, in some sense, those are normal activities performed over data; however, those activities can be performed in a malicious way. As examples, in a banking system a balance may be changed fraudulently, or in a car insurance company the number of at fault accidents of a customer may be deleted in order to diminish the payments.

Availability, integrity and confidentiality are the three most important requirements for handling data (1). Data must be present when the authorized user or owner of the information needs it. Data must be able to be changed in authorized ways by the authorized user or owner of the information, and finally, data must be known and changed only by the authorized users and owners of the data.

*1) Intrusion Detection Systems:* An *intrusion* to a computer is an action focused in gaining unauthorized access to the resources of the computer (2); that is the focus of a virus that can be embedded in an e-mail, a back-door in a computer program, an unauthorized user who subverts the login security of the computer, or an authorized user who tries to exceed his or her own privileges.

| *Year* | # Host(miles) | *# Vul.* | *# Incidents* | *Inc./Vul.* |
|---|---|---|---|---|
| 1995 | 4,852 | 171 | 2,412 | 14.11 |
| 1996 | 9,472 | 345 | 2,573 | 7.46 |
| 1997 | 16,146 | 311 | 2,134 | 6.48 |
| 1998 | 29,670 | 262 | 3,734 | 14.25 |
| 1999 | 43,230 | 417 | 9,859 | 23.64 |
| 2000 | 72,398 | 1,090 | 21,756 | 19.96 |
| 2001 | 109,574 | 2,437 | 52,658 | 21.61 |
| 2002 | 147,344 | 4,129 | 82,094 | 19.88 |
| 2003 | 171,638 | 3,784 | 137,529 | 36.34 |
| 2004 | 233,101 | 3780 | - | - |
| 2005 | 317,646 | 5990 | - | - |

TABLE I
NUMBER OF HOST, VULNERABILITIES, INCIDENTS AND INCIDENTS/VULNERABILITIES PER YEAR (DATA TAKEN FROM (3) AND (4)).

An intruder can exploit *system's vulnerabilities*, defined those as leaks in systems, like the ones found in Microsoft Internet Explorer 5.0, Netscape Enterprise Server 3.6, Real Secure Network Detection Intrusion Detection Software, and so on (5).

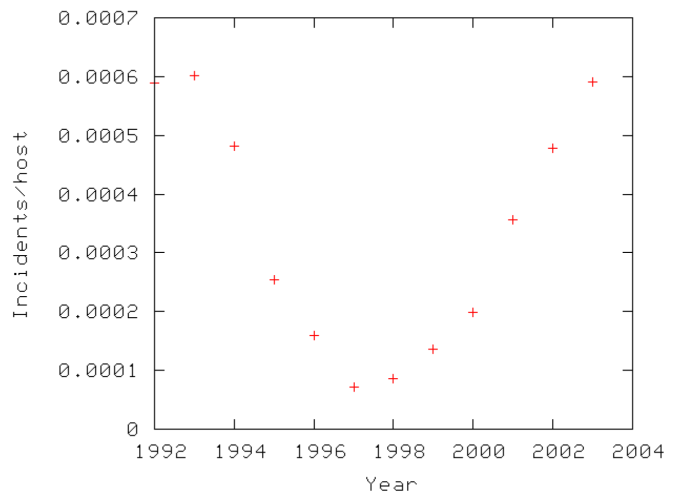


Fig. 1. Incidents Reported per Host Advertised.

Table I shows the the number of vulnerabilities reported since 1995 to the Computer Emergency Response Team (4). We can observe, that since 1998, each year the number of vulnerabilities reported[1] doubles the previous year—except 2004. The same Table I reports the number of incidents[2], and the incidents per vulnerabilities ratio. Looking this information and Figure 1—that illustrates the ratio of incidents per host advertised—we can appreciate the exponential grow of incidents, the dramatical increase on incidents per vulnerability and the increase of the ratio of incidents per host advertised since 1998, picture that gives an indication of the importance of security mechanism as intrusion detection systems.

An *Intrusion Detection System* (IDS) is a security mechanism that looks for intrusions. To do that an IDS must know in advance what the intrusions are—known as *misuse detection*— or it must distinguish between normal and abnormal activity in order to recognize intrusions (2).

A real-time IDS looks for audit trail records as they are generated by the operating system. By contrast, an off-line IDS performs analysis after the fact on audit trail files, i.e., it looks for past actions and reports intrusions that have already happened. IDSs usually have three modules (1):

1) the audit trail record collection module, or the corresponding information module,
2) the record analysis module, and
3) the results module that stores results and/or determines the action to be performed by the IDS, according to the security policy.

This paper is going to analyze Denning's intrusion detection model and two examples of IDSs, one that uses genetic programming and one that uses genetic algorithms.

## B. Evolutionary Computation

*Evolutionary computation* is inspired in biological evolution proposed by Darwin. In biological evolution species that adapt to their environment have a great chance to survive and reproduce through natural selection. Species that survive, usually develop new trends and capacities that can be inherited or learned by offsprings, if those are proved to be worth so that can be maintained through generations (6). In Evolutionary computation, a population of individuals survive and reproduce through "artificial" selection. Fittest individuals are selected to go to the next generation, or are selected to mate and possible crossover to give origin to offsprings. Little changes can occur in the phenotype of individuals that are called mutations and that can continue through generations if they prove to be "good" in maintaining the fittest (6).

In biology *Organisms* are composed of cells, *cells* are composed of chromosomes, *chromosomes* are composed of genes, and *genes* are the initial piece of information of live (7). The set of genetic material of an organism is called *genome*. A particular subset of genes is called *genotype*, and the particular "values" of genes in the genotype is called *phenotype*. In evolutionary computation, artificial genomes are evaluated "judging phenotypical expressions of genotypes" (6). The fittest are selected to crossover and mutate to give rise to possible a new genotype that is expected to be better than its parents. This artificial evolutionary process gives the idea of evolution to the best, i.e., climbing to a maximum.

Evolutionary computation has been used then, as heuristic method to solve approximately optimization problems (6). There are some optimization problems that can not be solved exactly because they are NP-hard or the computational resources to solve them exactly are quite expensive. Classical

[1]Should be noted *reported*, in the sense, that nobody knows how many was not reported

[2]Data available until 2003.

examples of NP-problems are the travel sales problem and the one-zero problem—which a classical example is shown in Section IV of this paper.

Evolutionary computation has been widely in artificial intelligence and as an optimization tool(6), applied in robotics, pattern recognition (8) and so forth. For a good understanding and mathematical justification of evolutionary computation see (6).

*1) Genetic Programming: Genetic programming* (GP) was introduced by John Koza with the purpose to generate automatically programs that solve approximate problems (8).

Genetic programming has been used widely in evolving neural networks, in complex systems like Lindenmayer systems and cellular automata, in evolving hardware, evolving parallel programs, and so forth (9).

Genetic programs evolved are Lisp-like programs, i.e., tree structures that encode programs using particular primitives according to the problem to be solved. the macro algorithm that is executed in order to generated those programs is (9):

```
(1) Random Generation of the
First Population of programs
(2)Iterate
        (2.1) Execute each program in the
        population and evaluate its output
        according with the fitness function,
        (2.2) Selection of best evaluated
        parents
        (2.3) Reproduction with crossover
        and mutation
Until Termination criterion has been satisfied
(3) The best program of the population according
with the fitness function is the approximate
solution.
```

As an illustrative example presented by (7), let us see briefly the generation of a program to calculate the orbital period *P* of a planet given its average distance *A* from the sun. The candidates to be nonterminals in the tree structure are the arithmetic operators and the terminal—in this case—will be the average distance from the sun of the corresponding planet *A*.

If we follow the last macro algorithm the first question is how many random programs are we going to generate in the first population. The answer is that that depends on the problem we are solving. For this particular example (7) presented three of those initially generated programs, see Figure 2.

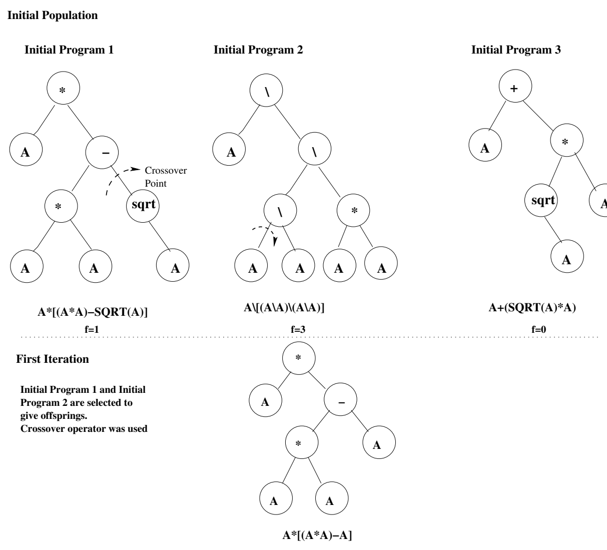


Fig. 2. Evolution of Programs. Taken from Mitchell M. (1998)

The fitness values assigned to these three programs are $f = 1$ for the first, $f = 3$ for the second, and $f = 0$ for the third. These fitness values were obtained comparing the program output *P* with six planet known orbits giving a $20\%$ of tolerance (7). This means, that if a program is approximately correct, its fitness value is $6$. We observe in Figure 2 that the best program was program $2$ and the worst was program $3$.

Continue with the macro algorithm, the best parents are selected—according with the fitness value, giving preference to the fittest—in this case parent $2$ and parent $1$, they crossover probabilistically and sometimes mutate to give origin to a offspring that is going to form part of the next generation. The process continues until an acceptable solution is found.

The genetic programming approach is quite rich in theory, for more details see (10), (11), (12).

## C. Genetic Algorithms

A *genetic algorithm* (GA) is an heuristic tool used to solve approximately hard problems. In genetic algorithms the possible solution is usually encoded as a bit string that is called chromosome. At the beginning of the evolution process a set of possible solution are generated randomly. This set of possible solutions is called *population*. The macro algorithm for genetic algorithm is similar as the one shown in previous section for genetic programming—see Section I-B.1.

As is common in evolutionary computation in GAs The evolution is guide by a fitness function, that is the one that evaluates each individual in the population, given better score to the fittest. Crossover and mutation operators are also used as is illustrated in Figure 3

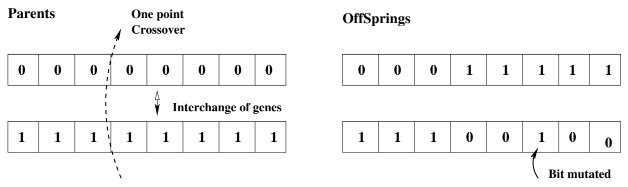


Fig. 3. Genetic Algorithms Crossover and Mutation Operators.

What are then some of the difference between GPs and GAs? Well in GP we have structure in each program in the population, and each program can dynamically grow or ungrow, i.e., usually there is no specific size. By contrary, as is usual in GAs, the chromosome has a fixed length [3]. In GP as the evolved program approaches the solution its fitness value approaches to $0$. In GAs as the chromosome evolve through the solution, its fitness value approaches a maximum. In GPs the syntax of programs should be preserved, i.e., a generated program should be syntactically correct. In GAs, there is no syntax, the problem is usually encoded as a bit string.

For more details and applications on GAs see (7), (13), and (6).

## II. An Intrusion Detection Model

Denning's pioneering work in IDSs is based on the assumption that an intrusion can occur if there is a deviation from normal activity (14). Normal activity here is the activity performed by an authorized user using the computer—the programs that are executed, the number of logins per day, and so forth—and the activity of the computer itself—the CPU time used, the amount of I/O activity, and so forth. See Figure 4 as an abstraction of static rule base system.

According to Denning, various types of intrusions can be detected with this model:

- masquerading: a user that is supplanted by an intruder, in this case, it is assumed that the intruder has a different activity pattern than the normal user,

[3]There are some GAs that use dynamic size in chromosomes too.

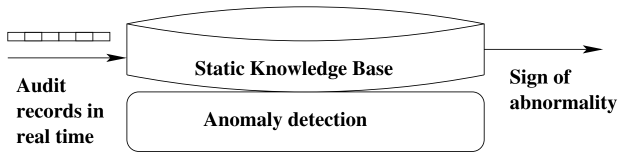


Fig. 4. Abstraction of Denning's Model.

- trojan horses: a program that is implanted by an intruder; here the response time of the computer can be degraded,
- virus: a program that is embedded in another program and replicates itself, usually using CPU time or performing a lot of I/O, and then the use of resources is abnormal,
- break-ins: when an intruder bypasses the security of the computer and gains access to it through the use of a "guest" account, supplanting a defined user in the computer, using a trojan horse, and so forth; in this case there may be more use of CPU or different patterns of the use of the system by the supplanted user,
- internal intrusions: a user authorized to use the system who tries to gain access to confidential information, or as a matter of example, he or she sends information to an entity not authorized by the company for which he or she works; in this case the legitimate user can direct data to a remote printer that is not used for that purpose.

However, there is the possibility of false alarms, i.e., there could be a deviation from the normal activity due to different causes than an intrusion, like the change of duties of an employee, the completion of a program, or a new program in production.

### *A. Components of Denning's Model*

As mentioned previously, an IDS controls the activity over the data and resources in a computer. There are users—*subjects*—that perform actions on data—*objects*. The actions performed by subjects on objects are recorded in profiles, in order to find deviations from normal activity. *Profiles* are, then, records that store the activity in the computer, according to the policy defined. A profile is uniquely identified by *name*, *subject*, and *object*.

*1) Statistical Models:* In order to capture intrusions the IDS has rules that use the following statistical models (14):

- The *operational model* where exceeding a predefined threshold is an indication of an intrusion. The threshold is normally defined by the security policy. For example, the security policy can establish that more than three failed attempts to enter a password should be reported.
- The *mean and standard deviation model* where a deviation from $mean \pm threshold * stdev$ is an indication of intrusion. The threshold in this case is different than the previous one, in the sense that this time *four* is normally used, because in a normal distribution almost $100\%$ should be in that interval.
- The *multi-variable model* where correlation between activities is used. For example, we may need to take into account CPU time along with I/O used by a program. It could be that looking at the use of CPU time alone is not enough for detecting an intrusion.
- The *Markov chain model* where activity is seen as events and the probability of occurrence of an event depends on the history. For example, if a programmer usually uses a set of commands in order to edit, compile, link and execute an application, then almost always the same set of commands is expected, and so the IDS can know what command is expected. If an unexpected command happens, then there is a suspicion of intrusion and the IDS will signal an alarm.
- The *time series model* where the time of occurrence of the activity shows its normality. For example, the time of running a banking settlement. The event occurs every business day at approximately the same time. If the activity happens at a quite different time (or has already occurred) then there is a suspicion of abnormal activity.

Besides statistical models, a metric is used. A *metric* is a statistical variable that corresponds to a new observation, i.e., a new audit record has been generated and so a value of the metric must be calculated according with the statistical model.

*2) Profiles:* In order to perform intrusion detection, Denning proposes the following data structure for a profile: *name*, *subject*, *object*, *action-pattern*, *exception-pattern*, *resource-usage-pattern*, *period*, *variable-type*, *threshold*, and *value*.

*Name*, *subject* and *object* form the key of the profile. *Action-pattern* is the action we are looking at as, for example, "read" or "delete." *Exception-pattern* matches the return field of the audit record, usually $0$ means success in the system call performed while another value indicates an exception, usually $-1$, for example "no such file or directory." *Resource-usage-pattern* corresponds to the usage of a resource such as CPU time or number of pages printed. *Period* corresponds to a time interval. *Variable-type* is the model used, i.e., operational, mean and standard deviation model, etc. *Threshold* is used in conjunction with the operational model and the mean and standard deviation model; and *value* corresponds to the actual value of the variable in observation.

Profiles are classified as *activity profiles*, *template profiles* and *anomaly profiles*. When an audit record is generated, the IDS actualizes the corresponding activity profile and, depending on the model and value, it can generate an anomaly profile and raise an alarm. If the activity profile does not exist, then it is created using a profile template.

One of the difficult parts of IDSs is the creation or initialization of profiles. Denning suggests in this case the use of templates. A template is a data structure with the fields as enumerated previously. If a new user is defined in the system, when the user performs his or her first login, the IDS is not going to find activity profiles, and then the IDS is going to generate the ones needed, using the corresponding template profiles. All fields of the template are copied to the new activity profile, except *subject*.

Profiles can be used by an individual subject or by an aggregation of them. In the same way profiles can be used by an individual object or by aggregations of them. For example, we can have a profile for subject "diaz" with object "passwd-file" and/or we can have a profile for subject "robotics" with object "passwd-file," where "robotics" is an aggregation of subjects.

## B. Analysis of Denning's Model

The motivation for adding IDSs to computers is addressed by Denning principally with regard to two facts: the need to cover known and unknown flaws in systems. However, her model does not completely accomplish that goal, in the sense that if a flaw is already known, the model presented does not cover known attacks or exploitation of that particular flaw.

The model, then, can be complemented with a misuse detection model, and this can be done in the following way: after the IDS looks for abnormal activity, then it could look for misuse, using the information stored in profiles. The system must have knowledge of the flaws or misuse in advance. This type of analysis can cover more than one profile depending on the flaw exploitation to be detected, and should be done after the profiles have been updated. For the case of an unknown flaw, it is almost impossible to predetermine what type of user activity will be related to that flaw, in order to create the specific rule to protect against it.

Denning's model is based in what constitutes "normal activity." A particular user who is not intruding can still deviate from an expected pattern of normal activity according to the IDS, if we are using, for example, a mean and standard deviation model. Denning proposes to solve this problem with the use of aggregate profiles, called *classes of profiles*, so activity of users can be compared with patterns established by a class. As an example, consider the mean elapsed time between login and logout of a user and the corresponding value for an aggregation of users. If the value of the user deviates beyond the threshold of the aggregation, an alarm can be raised. It must be taken into account that the metric of the aggregation—following this example—is calculated as the mean of the total frequencies of the individual user profiles that belong to that class (14). This approach can be useful if the activity of the user is low too, because there is going to be a deviation from the norm in this case as well. This assumption, of intrusion as deviation from normal activity, is a good start to build an IDS. However, the problem of differentiating between "normal" and "abnormal" can be huge, if we consider the type of activity carried out by different employees in a company.

Another aspect to consider is the initialization of profiles, i.e., template profiles, because what can happen is that a new user may not be familiar with the system and so the number of false alarms can be high (1). Denning proposes a flexible or high threshold at the beginning. However, a malicious user can deviate from the expected normal activity not only from the start, but also at whatever time; in the mean and standard deviation model, for example, the mean and standard deviation can be moving slowly toward an intrusion. If we compare the user profiles with the ones of his or her class—as suggested by Denning—there is no guarantee of the discrimination of the entire class compared with the particular user class. For example, in the class there may be two users, and the one that is performing malicious activity may be the one who performs more activity in the system. The class itself should be weighted enough, i.e., if there are not "enough" users in the class, then the class itself can be shifted by the activity of the individual user, and the approach to the problem then is useless. A complementary approach—that we suggest—is to take into consideration how the mean for a profile changes in specific epochs of time (monthly, for example). If the mean has a tendency to increase or decrease, then some abnormality can be present. As another approach, two complementary models can be used to solve this problem—the operational model and the mean and standard deviation model—if the mean goes far a threshold then an alarm can be set, indicating the presence of a possible deviation from normality.

Another difficulty in the same line is related to the difference in activity at different times or periods of the day or days. As an example, the activity performed during a weekend is usually different from other days, or the activity performed during the hours of the day is different from that performed during the night (1). Denning suggests having table structures with the specific metric and model according to the hour or day. This solution seams reasonable; however, an excellent knowledge of each type of user profile is needed in order to disaggregate events by months, days and specific hours, not to say the huge amount of work to be done for all employees in a company, in order to accomplish this task. A complementary suggestion here—given by us—is to have *sentinel profiles* for those days and hours of almost no activity. If, for example, an employee uses the system Monday through Friday, from 8:00 am to 5:00 pm, then we can use the users' profiles for those days and hours, and the sentinel profiles are used for the other days and hours (weekend and night hours). A sentinel profile would be a profile with its threshold zero or near zero, according to the policy. If there is more activity than the near to zero threshold, then an alarm of possible intrusion can be raised.

How the system reports intrusions or how the system stops to report intrusions is something that is not addressed in Denning's document. If the IDS has a specific terminal where all messages are printed, then an alert can be lost among a huge number of messages—if such are present. The IDS then should wait for the reception of a signal from the security manager indicating that an action on the intrusion has been taken. This implies that the operator of the system is responsive 24/7, a fact that is important in the sense that the IDS can detect an intrusion but if no body takes care of it immediately, extensive damage can be done. Once a intrusion is detected and reported, how can the system be set to a normal state again? Is the profile activity that generates the alarm re-set? If that is not done, then the system will probably continue raising alarms—let us say, for example, that a password failure value was reached, and that the security policy says that after three failed attempts the user account is blocked. Then, first of all, the security manager or operator should check if there is an intrusion and handle it. After that, the account should be unblocked, if that is the case, then the activity profile should be reset, so the IDS does not continue alarming the system with the possible intrusion.

The IDS model proposed could be quite a burden for the system it monitors. If almost every audit trail record triggers an action in the IDS, then the entire system can be overloaded in terms of CPU time, principal memory, and/or disk space required. There is a trade-off between security and load on the system. Besides that, some intrusions—as suggested by Denning—can be difficult to detect with the model proposed. For example: it is almost impossible to control the leak of sensitive information. Denning says that this intrusion can be captured if data is routed to a remote printer not normally used but in the model there is no mention of device control, i.e., which types of devices are allowed to be used by users. The same happens with the inference of data. Denning says that a deviation will be noted if the user retrieves more records than usual but the fact is that the user could retrieve a normal number of records and store that information in order to do aggregation later. For the case of viruses, if there is no anti-virus in the system, a virus can delete important information before it is detected, i.e., it is not always the case that a virus uses significant system resources—and can so be detected. The goal of the virus can be to destroy or damage important parts of the system. A denial of service could not be detected either. The denial of service can block

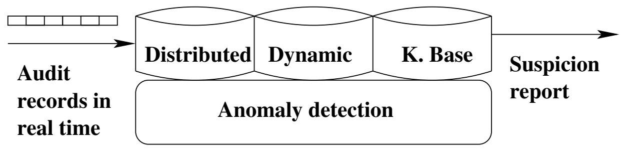


Fig. 5. Intelligent and Dynamic Agents Doing Anomaly Detection.

the entire system, including the IDS itself.

If the IDS model proposed is centralized, then it can offer a central point of attack for an intruder. Besides, if the IDS crashes then the entire system is going to be unprotected. The next section is going to present a distributed architecture that avoids this particular issue.

## III. A Genetic Programming Approach

One approach quite close to Denning's ideas was proposed by (15). Their IDS is based on Denning's premise that intrusions can be detected due to deviations from normal activity. They implemented the IDS using two ideas: genetic programming and agents.

### *A. Components of the Genetic Programming Approach*

The genetic programming approach is placed in a rule base system. Agents learn normal and abnormal activity using a feedback-learning paradigm. Each agent is a program specialized in a particular type of intrusion. Agents act independently in a dynamic environment—the changing environment of the computer system (15). See Figure 5 as an abstraction of this dynamic model.

The security of the computer is based on and supported by a *security policy* (15). The security policy guides and governs all the activities done by users in computers—what activity is good, what is bad, what is allowed, what is not allowed, how to protect the system from intrusions, and how to react in cases of intrusions. The security policy defines security standards and mechanisms in a computer system.

*1) The Rule Base System:* The question here is how are we going to capture abnormal activity? (15) propose that the computer itself, using artificial intelligence, find the rules.

How does genetic programming work? As with almost every program in a computer, it has an input, a process, and an output. The input in this case is the activity of a user in the computer, the process is the rule base generated with the learning paradigm, and the output is the suspicion grade of the input received.

In order to generate the rule base system, evolution is used. Programs evolve guided by a *fitness function* that scores the performance of each program evolved. Operators and operands are given to each program in order to solve the problem. Initially, a pool of programs can be generated randomly; each is evaluated according to the output given by it, in this case, the suspicion grade. If the program performed well, it receives strong positive feedback and, on the contrary, if the program evaluates the input incorrectly, it receives less positive feedback.

After each trial, the best programs are chosen in order to perform crossover and mutation, with some probability. The new programs generated are again presented with an input, and their output is evaluated again. If there is an intrusion and the program ranks it as such, then the program receives a good score and, on the contrary, if there is no intrusion and the program evaluates that as an intrusion, then the program receives a bad score.

Genetic programs have a tree structure, where inner nodes are operators and the leaves are operands. As a matter of example, let the operators be the set of arithmetic operators {+, -, *, /} and the operands the set of integers, then we can generate a set of programs that perform the basic arithmetic operations, as the example shown in Section I-B.1. However, programs generated are not in a standard computer language such as C or Java; a set of primitives to solve the particular problem is used. For example, `source-port`, `destination-port` (15), and `my-address` (16) which return integer values, and that can be used in conjunction with `generate-suspicion-broadcast` (16), the arithmetic and comparative operators, along with *FOR* and *IF THEN ELSE* clauses, can be used to form program sentences like (taken from Crosbie and Spafford):

```
for-each-packet-do
    if dest-port != my-address
       then
        gen-suspicion-broadcast
    end-if
end-for
```

Although Crosbie and Spafford highlight that primitives should be handled independently—for example, primitives to handle interconnection time should be independent from primitives related to port addresses—in order to preserve semantics, there could be some instances when the combination of some of them can be useful for doing intrusion detection, like the ones to handle `source-port` or `destination-port` combined with `average-interconn-time`:

```
for-each-packet-do
  if (0 < dest-port  < 1025)
     and (aver-interconn-time >
     520) then
        gen-suspicion-broadcast
    end-if
end-for
```

The output of each program—the suspicion value—is ranked using a fitness function. The fitness function is the one that guides the evolution, selecting the best programs from each generation.

The fitness function proposed by (15) is

$$F = (100 - \delta) - penalty$$

where $\delta = |outcome - suspicion|$, and $penalty = (\delta * ranking)/100$.

*Outcome* is a predefined value that the security manager of the system gives as the intrusion value. For example, $90\%$ is a high possibility of an intrusion, $20\%$ is a low possibility of an intrusion. *Suspicion* is the output of the program that is being evaluated. If the program correctly evaluated the input, then $\delta$ is near $0$ and $F$ is near $100$ (contrary to what was written by Crosbie and Spafford). If, on the contrary, the program does not correctly evaluate the input, then it is going to receive a penalty. The penalty is ranked depending on the difficulty of the scenario (17), i.e., the input. If the scenario is quite obvious, then the ranking is high; if, on the contrary, the scenario is somehow difficult to discriminate, then the ranking is low. In this way (17) tried to be fair with each program's suspicion value.

*2) Distributivity of the IDS:* The architecture of the IDS is distributed, i.e., different programs run in the target system, each looking to a specific type of intrusion. Each program sends its suspicion value to a centralized control system that gathers all the suspicion values of all the agents and, depending on those values, can set an alert.

Agents can be added and/or removed, giving flexibility to the IDS. Agents can be retrained in order to capture new types of intrusions (15).

Agents work cooperatively. As an example, one agent can be looking at the number of connections to a specific port, the duration of each connection, the minimum and the maximum duration time of each connection, the source, and the destination port. Other agents can be looking at specific requests to a Network File System (NFS), and so forth. The first agent can raise its suspicion value to a specific connection that has taken place, and the NFS agent can raise its suspicion value to a write request, for example. Both send their suspicion values to a MUX, which evaluates the danger and sends an alert to the security manager, if required by the policy.

*3) Tests and Results of the Agent System:* After generating the programs using genetic programming, the three best programs were chosen and were put in a production environment, where all were submitted to different scenarios presented in Table II (redrawn from Crosbie and Spafford, 1994). The outcome of each scenario is given by the expert and indicates his or her evaluation of the scenario as being an intrusion or not. The suspicion value reported for each agent is summarized in Table III (redrawn from Crosbie and Spafford, 1994).

As can be seen in Table III, *connections to privileged ports* was well classified by all the agents. However, *Login then long pause then logins* that corresponds with *Rapid connections, then random pauses* in Table II with a high outcome ($80\%$) was misclassified by all the agents. The third agent's suspicion was $0\%$—the worst. The last scenario, which was not intrusive, was severy misclassified by Agent 1, somewhat misclassified by Agent 2, and even slightly misclassified by Agent 3.

### B. Analysis of the Approach

The idea behind the use of multiple specialized autonomous agents gives advantages to the building of IDSs:

- *Scalability.* Agents can be added and removed from the system dynamically. New types of intrusions can be monitored without disturbing the distributed system, by the addition of new agents as needed (17).
- *No single point of vulnerability.* The distributed architecture of the approach gives no single point of failure for the IDS. However, an agent can be disabled from the system and there is no reference to how the system manages that situation. The net effect is that the IDS will have a hole. As Crosbie and Spafford suggest, an upper MUX combines suspicion reports from all the agents; we can suggest that to solve this problem, if the upper MUX has not heard from an agent during a specific period of time, then the Upper MUX should query that agent in order to figure out if it is still running.
- *Specialization of each agent.* Each agent is trained to accomplish a specific goal. This gives granularity to the development of the system; maybe the first agents can be developed to capture the principal intrusions and later on others can be developed that can capture less critical intrusions.
- *The use of artificial intelligence.* The rule base system is proposed as automatically generated by the computer, so according to (15) there is no need for an expert to write the rule base system. However, there should be an expert in order to develop the training scenarios and give the outcome of each one.

Some deficiencies are also associated with this approach:

- *There is no mention of control.* If an agent is killed, the IDS should report the event and might restore to the killed process. They mention (17) an upper MUX— that combines suspicion reports—that receives each agent's suspicion value, combines them into an overall suspicion report and passes it to the user, but there is no tracking of missing agents in the IDS. (See comments under "No single point of vulnerability" above.)
- *The architecture inherits the deficiencies of statistical IDSs.* Although artificial intelligence is used to find parameters and thresholds, more tests should be done in order to figure out the presence of a high percentage of false alarms.
- *Previous knowledge of the outcome for the specific scenario is implied.* So, what happens if a new scenario arises? What would be its outcome?
- *There is no mention on how the suspicion value is calculated.* The authors highlight that each agent generates the suspicion value ac-

| Type of Scenario | Outcome |
|---|---|
| 10 connections with 1 second delay | 90% |
| 10 connections with 5 second delay | 70% |
| 10 connections with 30 second delay | 40% |
| 10 connections every minute | 30% |
| Rapid connections, then random pauses | 80% |
| Intermittent connections | 10% |
| Connections to privileged ports | 90% |
| Connections to any port | 70% |

TABLE II

TRAINING SCENARIOS

| Activities | Suspicion report | | |
|---|---|---|---|
| | A1 | A2 | A3 |
| Connections to privileged port | 83% | 100% | 98% |
| Login then long pause, then logins | 31% | 26% | 0% |
| Logins and FTP with long pauses | 73% | 47% | 25% |

TABLE III

TEST CASES AND AGENTS' REPORTS

cording with the scenario presented—difficult or easy—but how is that value calculated? E.g., how does the system determine whether the agent gives 80 for suspicion value or 83? It is inferred that there exists a correlation between the ranking of the scenario and the suspicion value but the authors do not give such a correlation—at least in the documentation referenced until now.

- *There is no mention of how to choose the best agent to put in production*. The authors propose that after the training step, the best agent is put in production but with the example provided—see Table IV—the three agents perform well in some scenarios and bad in others, so finally, which is the best? One possible suggestion would be to take into account the false positive and false negative rate of each one and choose the one with best ratios—performing, of course, more tests.
- *There is no mention on how the upper MUX combines the suspicion values reported by each agent*. The authors propose that each agent reports its suspicion value to an upper MUX, which combines those values and gives a final verdict to the system administrator, but there is no information on what criteria is used by the MUX in order to decide, based on the suspicion values reported by each agent, if there is a possible intrusion or not.
- *The system is proposed as an anomaly detection system, but the agents are trained for misuse detection*. Anomalous is a deviation from "normal" behavior, this means that we have knowledge of what is normal and whatever deviates from normal is suspected to be an intrusion. (17) state that the training scenarios have a mixture of both intrusive and non-intrusive activities. The agents are reporting intrusions based on previous knowledge of intrusions.
- *The examples shown are quite simple.* A query can accomplish the goal of the agents presented as examples. For example, to determine if a port is privileged or to figure out if there is a suspicion average time between connections, can be handled by a query operation. Of course, this is because the granularity presented in this model is high. Maybe if an agent is trained to captures a virus, and a new type of virus is present, it is possible that the agent capture the new one, because of the generalization of a learning system, in this case from genetic programming.
- Noting Table III, the authors report that *Agents* 2 *and* 3 *performed better than agent* 1. However, calculating the fitness values for those agents and setting the parameter *ranking* with

| Activities | Type of scenario | Outcome | Suspicion report | | | Fitness value | | |
|---|---|---|---|---|---|---|---|---|
| | | | A1 | A2 | A3 | A1 | A2 | A3 |
| Conn. Priv. Port | Conn. Priv. Port | 90% | 83% | 100% | 98% | 89.5 | 85.0 | 88.0 |
| Login pause logins | Rapid conn. rand. pause | 80% | 31% | 26% | 0% | 26.5 | 19.0 | -20.0 |
| Logins and FTP | Intermittent conn. | 10% | 73% | 47% | 25% | 5.5 | 44.5 | 77.5 |

TABLE IV

FITNESS VALUES FOR THREE AGENTS.

a value of $50$—exactly the middle of 100[4]—it is difficult to say exactly which agent was the best, because we can argue that *Agent 1* performed better that Agents $2$ and $3$ for the first two scenarios as is shown in Table IV. We can change the ranking value of scenarios supposing it "difficult" (ranking = $10$) or "easy" (ranking = $90$) as and the result holds.

## IV. *GASSATA*, A GENETIC ALGORITHM THAT PERFORMS OFF-LINE INTRUSION DETECTION

*GASSATA* (18) is an off-line tool that increases security audit trail analysis efficiency. The main ideas of this algorithm are the following:

- to perform misuse detection by comparing the user's behavior against a matrix of known attacks,
- to explain the data contained in the audit trail by hypothesizing the occurrence of one or more attacks, and
- to use a heuristic method, genetic algorithms, to solve it because explaining the data is an NP-complete problem.

See Figure 6 as an abstraction of this model.

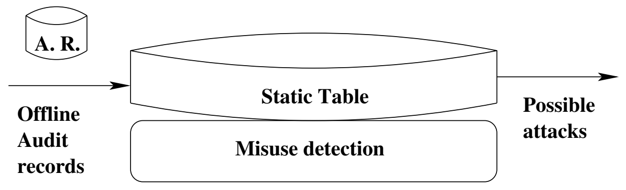


Fig. 6. Off-line Misuse Detection Abstraction.

### A. Description of the Genetic Algorithm Approach

This approach can be formalized thus:

- let $N_e$ be the number of distinct event types audited,
- let $N_a$ be the number of known attack types,
- let *AE* be an $N_e \times N_a$ *attack-event* matrix that gives the set of events generated by each attack,
- let *W* be a one-dimensional array of length $N_a$ called the *weighted vector* that gives the risk of each attack,
- let *OV* be a one-dimensional vector of length $N_e$ called the *observed vector*, where $OV_i$ counts the number of events of type $i$ present in the audit trail, and
- let *I* be a one dimensional vector of length $N_a$ called the *hypothesis vector*, where $I_i = 1$ if attack $i$ is present and $I_i = 0$ otherwise.

To explain the data contained in the audit trail by the occurrence of one or more attacks, *GASSATA* attempts to find the $I$ vector that maximizes the $W \cdot I$ product, i.e., maximizes

$$\sum_{i=1}^{N_a} W_i * I_i$$

subject to the constraint

$$(AE \cdot I)_i \leq OV_i$$

So, in order to evaluate the hypothesis $I$ corresponding to a particular subset of attacks, the algorithm first calculates the $W \cdot I$ inner product and then the constraint.

In order to evaluate the constraint, the algorithm counts the number of events of each type generated by all the attacks hypothesized in $I$. If these numbers are less than or equal to the number of events recorded in the audit trail, then the hypothesis is realistic. On the contrary, if some of those numbers are greater than the actual number of events that occurred, then the proposed solution $I$ is penalized as being unrealistic.

The fitness function proposed by Mé that captures these ideas is then

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

[4]Authors set the increment in penalty by $ranking/100$ but they do not provide ranges for *ranking*.

where $\alpha$ is used to maintain $F(I) > 0$, in order to maintain some diversity in the population, $\beta$ provides a slope for the penalty function, $P$ corresponds to the penalizing of a proposed solution that fails the constraint, i.e., if $(AE \cdot I)_i > OV_i$. The penalty function is

$$P = T^2$$

where $T$ is the number of times for which $(AE \cdot I)_i > OV_i$. Mé reports good results with GASSATA.

### *B. Analysis of the Fitness Function suggested in* GASSATA

As with many heuristic tools, we have here the problem of how to provide accurate values for parameters, in this case $\alpha$, $W$, and $\beta$ in order to maximize the fitness function $F(I)$:

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

*1) Case I (19): $\sum_{i=1}^{N_a} I_i$ vs. $T^2$ comparison.:* As we are going to compare the term with *I* against the term with *T*, let us begin by setting parameters as $\alpha = 0$, $\beta = 1$ and $\forall i\ W_i = 1$[5], and take into account the variables involved in the fitness function. This approximation gives us

$$F(I) = \sum_{i=1}^{N_a} I_i - T^2 \qquad (2)$$

Testing was done with Equation 2, and it was found that all the individuals had fitness equal to zero in the first generation. The algorithm was run five times and in all the cases the fitness for each individual was zero.

The last fact seems to happen because the term $T^2$ tends to dominate the term $\sum_{i=1}^{N_a} I_i$. In fact, $0 < T^2 < N_e^2$, and $0 < \sum_{i=1}^{N_a} I_i < N_a$, where $N_e = 28$, and $N_a = 24$, for this study.

So now we take into account the use of parameters. There are at least three ways to handle this result in this particular case, in order to get positive fitness values:

1) set $\alpha$ to a sufficiently positive value,
2) set $\beta$ to a sufficiently small positive value less than one (i.e., $0 < \beta < 1$), and
3) use $T$ rather that $T^2$ (and use an appropriate $\beta$). In this case we are considering the power of $T$ as a parameter, and maybe it is quite high, so we suggest as an alternative to change it to $1$.

[5] $W_i$ was chosen equal to 1 $\forall i$ in order to simplify the analysis

Let's deal with the first two. As the average of $\sum_{i=1}^{N_a} I_i$ is $12$ (for random hypotheses) and the average $T$ is $14$ (i.e., $T^2$ is 196 using the average of $T$) then, in order to balance the two terms and have a $F(I)$ positive, we set $\beta$ to $1/20$, then Equation 2 becomes

$$F(I) = \sum_{i=1}^{N_a} I_i - (1/20) * T^2 \qquad (3)$$

This time an average $21\%$ of the individuals had fitness values less than or equal to zero in the first generation, so things were better, but not enough; we are trying to implement Mé's paper, and he proposes to use $\alpha$ in order to avoid a negative fitness value (18).

So, now it appears necessary to use $\alpha$ in order to handle the $21\%$ of individuals that have fitness less than or equal to zero. Testing was done in order to appreciate the magnitude of the negative value of these individuals, and the maximum found was $-4.0$. So, setting $\alpha$ to $4.0$, Equation 3 becomes

$$F(I) = 4.0 + \sum_{i=1}^{N_a} I_i - (1/20) * T^2 \qquad (4)$$

Table V shows the results obtained for user with $ID = 2051$—data downloaded from the Lincoln Laboratory—and the fitness function as in Equation 4. The column *Detected* means the number of intrusions detected by the algorithm—in this case, there are in total $4$ intrusions in the file.

There are a great number of false positives. So, is the penalty function not functioning? Is $\sum_{i=1}^{N_a} I_i$ guiding the algorithm in the wrong way? To examine these questions the next section looks at the third approach just proposed, i.e., the use of $T$ instead of $T^2$, comparing $\sum_{i=1}^{N_a} I_i$ with $T$.

*2) Case II (19): $\sum_{i=1}^{N_a} I_i$ vs. $T$ comparison.:* Let us now try the third case where the penalty function is set to $P = T$, and in order to do analysis let us combine the two terms $\sum_{i=1}^{N_a} I_i$ and $T$ into the fitness function

| Run | False + | False - | Detected |
|---|---|---|---|
| 1 | 10 | 1 | 3 |
| 2 | 7 | 0 | 4 |
| 3 | 8 | 1 | 3 |
| 4 | 9 | 1 | 3 |
| 5 | 9 | 0 | 4 |
| 6 | 9 | 1 | 3 |
| 7 | 8 | 1 | 3 |
| 8 | 10 | 1 | 3 |
| 9 | 10 | 1 | 3 |
| 10 | 7 | 1 | 3 |

TABLE V
RESULTS WITH $F(I) = 4.0 + \sum_{i=1}^{N_a} I_i - (1/20) * T^2$

$$F(I) = \sum_{i=1}^{N_a} I_i - T \qquad (5)$$

We still observed some negative values and, as $\sum_{i=1}^{N_a} I_i$ and $T$ are quite similar in this test, we make use of the $\alpha$ parameter. We found that the greatest negative fitness value was $-4.0$ again, so we set $\alpha = 4.0$. The results are shown in Table VI for the same user as in Case I.

| Run | False + | False - | Detected |
|---|---|---|---|
| 1 | 7 | 0 | 4 |
| 2 | 7 | 1 | 3 |
| 3 | 12 | 0 | 4 |
| 4 | 5 | 0 | 4 |
| 5 | 6 | 1 | 3 |
| 6 | 5 | 0 | 4 |
| 7 | 7 | 1 | 3 |
| 8 | 6 | 0 | 4 |
| 9 | 8 | 0 | 4 |
| 10 | 4 | 0 | 4 |

TABLE VI
RESULTS WITH $F(I) = 4.0 + \sum_{i=1}^{N_a} I_i - T$

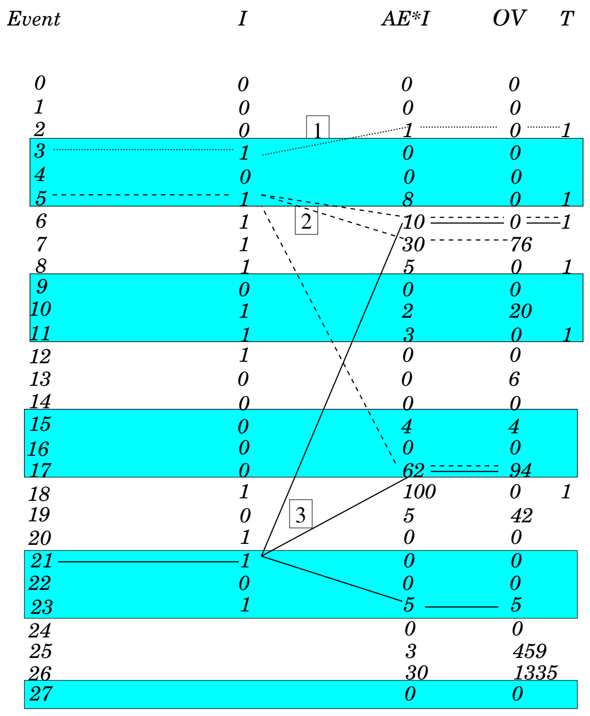
Now, using Equation 5 instead of Equation 4 we can address the following facts:

- there are $30.9\%$ fewer false positive,
- there are $62.5\%$ fewer false negative, and
- the algorithm found seven of ten times the total number of intrusions (there were 4 intrusions), against two of ten times, as was shown in Tables V and VI.

However a *great number of false positives* is still present (see Table VI). What is causing that problem? The answer is that the role of the term $\sum_{i=1}^{N_a} W_i * I_i$ is to guide the solution to have the maximum number of intrusions. However, this is good enough *only 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. The next section shows in detail how fitness values are calculated.

*a) Fitness function evaluation.:* Table VII gives the attack event matrix *AE* (18), an individual $I$ hypothesized in the last generation, the $AE \cdot I$ product, the observed vector *OV*, and the counts of overestimates for that individual $T$.

How does the fitness function accomplish its goal? Each 1 in the $I$ vector adds one to the fitness value, and the total of overestimates elevated to the power of two is subtracted according to Equation 1.



| Event | I | AE*I | OV | T |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 0 | |
| 4 | 0 | 0 | 0 | |
| 5 | 1 | 8 | 0 | 1 |
| 6 | 1 | 10 | 0 | 1 |
| 7 | 1 | 30 | 76 | |
| 8 | 1 | 5 | 0 | 1 |
| 9 | 0 | 0 | 0 | |
| 10 | 1 | 2 | 20 | |
| 11 | 1 | 3 | 0 | 1 |
| 12 | 1 | 0 | 0 | |
| 13 | 0 | 0 | 6 | |
| 14 | 0 | 0 | 0 | |
| 15 | 0 | 4 | 4 | |
| 16 | 0 | 0 | 0 | |
| 17 | 0 | 62 | 94 | |
| 18 | 1 | 100 | 0 | 1 |
| 19 | 0 | 5 | 42 | |
| 20 | 1 | 0 | 0 | |
| 21 | 1 | 0 | 0 | |
| 22 | 0 | 0 | 0 | |
| 23 | 1 | 5 | 5 | |
| 24 | | 0 | 0 | |
| 25 | | 3 | 459 | |
| 26 | | 30 | 1335 | |
| 27 | | 0 | 0 | |

Fig. 7. Fitness function evaluation

Figure 7 shows three cases (20). In $\boxed{1}$ we have a first case. An intrusion of type 3 was hypothesized ($I_3 = 1$) so 1 more is added to the fitness value and, as the multiplied vector that corresponds to this entry gives more events than the number of events that really happened ($MV_2 > OV_2$), one is subtracted from the fitness. So, one is added, because the attack was hypothesized, and one is subtracted, because the hypothesis was wrong, corresponding to entry two in the $(AE \cdot I)_i < OV_i$ comparison. The total change to the fitness is zero, making no difference

**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 VII

*AE* MATRIX TAKEN FROM LUDOVIC MÉ, I VECTOR, $AE * I$ PRODUCT, OBSERVED VECTOR *OV*, AND PENALTY COUNTS *T* AND *T'*

in the fitness value despite an incorrect hypothesis. This allows for false positives.

In $\boxed{2}$ we have a second case. An intrusion of type $5$ was hypothesized so 1 more is added and this intrusion is influenced by events $6$, $7$, and $17$ (see Table VII). In the case that an attack hypothesis requires a number of events greater than the number of events that really happened, one is subtracted. This is the case for event $6$ where $10 > 0$. For event $7$ there is no penalty—we observe $30 < 76$, and for event $17$, there is no penalty either because $62 < 94$. Then one was added because the attack was hypothesized, and one was subtracted, because for event type $6$ there were more events required than really happened. So in this case—as the previous one—there is no difference in the fitness value despite the fact that the algorithm produced a false hypothesis.

In $\boxed{3}$ we have a third special case. An intrusion of type $21$ was hypothesized and this entry is influenced by three positions in the hypothesis vector: position $6$ **again**, $17$ **again**, and $23$ (see Table VII). So, in this case, again, as an intrusion has been hypothesized one is added to the fitness value. As for the penalty, for the cases of positions $17$ and $23$ there is no penalty at all because the corresponding entries for $17$ and $23$ in the multiplied vector are less than the corresponding entries in the vector of events that occurred, and for the case of event $6$, there is no penalty either, because the penalty was **already** taken into account for the hypothesized intrusion of type $5$ (case $\boxed{2}$). So in this third case, where there should be a penalty, there is no penalty, and **this makes the algorithm get false positives too**.

So cases $\boxed{2}$ and $\boxed{3}$—that correspond to intrusions $5$ and $21$—have a common event to check—event type $6$—and this common event should be taken into account in the calculation of the penalty, i.e., if two violations occur—as was described previously—then the penalty value should be $2$, instead of $1$ as was implied by (18).

More testing was done in order to corroborate the false positives obtained when running the algorithm with the fitness originally suggested. The algorithm then fails with the fitness function proposed and the parameters used (21).

*3) A New Fitness Function is Proposed:* As stated before, the term $\sum_{i=1}^{N_a} I_i$ was incorrectly guiding the fitness function, and the term $T^2$ was counting the over estimates in an incorrect way. So the solution proposed has two parts (22) (20) (19):

1) remove the positive component $\sum_{i=1}^{N_a} I_i$, and
2) count overestimates in a correct way; this means, if two intrusions require an excess number of occurrences of the same event then count them twice, and so forth.

With this in mind and the experience gained with testing, the fitness function proposed only has the penalty function, and as the number of events is $N_e$, the new fitness function suggested is

$$F(I) = N_e - T'$$

It must be taken into account that the role of $\alpha$ corresponds now to $N_e$ and that $\beta$ is equal to one. However, the term $\sum_{i=1}^{N_a} I_i$ was removed, as stated before, in order to avoid false positives.

Another way to justify removing the positive component 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, $F(I) \rightarrow N_e$, the maximum in the function proposed. The fitness function is evaluating only the $T'$ term. There is a maximum when $T' = 0$.

| User | 10 run Average | | | % | | |
|---|---|---|---|---|---|---|
| | 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 VIII
RESULTS WITH $F(I) = N_e - \sum_{i=1}^{N_e} T_i$

It must be stated that this fitness function was found after much testing and that is not the goal of this report to show all the steps followed. But, in order to give some insight into the way this fitness function came into account, note that a two phase algorithm was considered: one phase determines the *number* of intrusions, and another determines the intrusions themselves. The corresponding fitness functions proposed were: $\sum_{i=1}^{N_a} I_i - N_a/N_e * T$ for *phase I*, and $N_e - T$ for *phase II*. Problems were found again with the fitness function $\sum_{i=1}^{N_a} I_i - N_a/N_e * T$, because of the positive side. Then this phase was abandoned and phase II was adopted. Phase II requires the number of intrusions to be found and, as it is not known, this parameter is set to the maximum number of intrusion—in this paper 24. Phase II begins to climb in order to find the $N_a$ intrusions and, in doing that, the algorithm sequesters intrusions until it can not climb more. In this case, the ones that are sequestered correspond to local maximums—fitness value of $N_e$—, i.e., individuals with intrusions. So, each time a new local maximum is found it is compared with the previous one, intrusion by intrusion; if there is a new one, then the entire set is tested again in order to check if there is violation of the constraint; if there is no violation then this new intrusion is added, and if there is a violation, the new intrusion is cataloged as an exclusive intrusion.

The results found with the new fitness function are shown in Table VIII. As can be seen, with the new fitness function *there are no false positives* and the number of *false negatives decreases dramatically*. This time 70 runs were performed with different data (some data were downloaded from

MIT's Lincoln Laboratory (23)) and only one time a false negative was present.

*4) Improvements to* GASSATA*:* Logs are part of systems and security logs are necessary in critical applications, especially those related with secrets of states, customer databases, and bank transactions, to list a few. Almost all computer systems have logs; the problem is the use of the log's data, because usually there is no adequate tool to read them; sometimes the computer administrator has to read the log as a bunch of text-based information.

In this research a tool is developed to read logs, with the principal purpose to get intrusions, following the guidelines of a genetic algorithm, *GASSATA*, suggested by (18), and improves that algorithm 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, and
- detect the absence of intrusions.

## V. Conclusions and Future Work

The need for automated audit trail analysis in computers, as reported by (24) is still present. The use of information provided by operating systems adds capabilities to the security of the information and data in computers. With IDSs that use logs containing events, file access and user activity can be monitored. Any time a specific file is opened or closed, or a login or log off procedure is executed, they are logged in the audit trail file, and the IDS can monitor this, based on its current thresholds.

Intrusion detection is an integral part of computer security. Intrusion detection improves the security of information systems by allowing the review of patterns of access in order to discover abnormal activity of users and serving as a deterrent to users' attempts to bypass system privilege or protection mechanisms.

Although many approaches have been explored for doing intrusion detection, most commercial products confine themselves to traditional methods like statistical characterization of user activity and usage patterns (1), as Denning's seminal work proposes. Although her model is based on the assumption of what constitutes "normality," her model could be complemented with misuse detection. These two types of intrusion detection—abnormal and misuse—constitute the two sides of a coin. With abnormal deviation unknown intrusions can be discovered and with misuse detection known intrusions can be recognized. Moreover, the possible overload imposed by the use of profiles can be reduced with risk analysis, maintenance, and distributed of the IDS. With risk analysis high risks should be evaluated and those could be addressed with a distributed architecture. Maintenance closes the cycle of user activity in the system, i.e., people who should no longer have access to the system have their privileges removed.

Research, study, and improvements of new intrusion techniques is a challenge that is addressed principally by the university community. This report presented some of this new research in intrusion detection, in one case by the use of genetic programming (GP) as proposed by (17). Although their idea of use of agents is quite good, the consideration of control should be addressed, as well as the overhead imposed by the training of the agents and by the agents on the system. Although the papers reviewed showed simple rules to get intrusions, more research should be done with this paradigm. However, a question that arises is: are there other heuristic methods that can accomplish this task better than the GP approach proposed? If we compare with another heuristic method like a genetic algorithm (GA)—as a matter of example—we see that GP is better, in the sense that it was designed to evolve solutions of different sizes, a matter that differs somehow from the common use of GAs as fixed length solutions. GP offers a chance to see intrusion detection systems with the ability to evolve. Agents can be retrained in order to get new types of intrusions. However, the period of retraining is an open question.

For the case of the GA as an analytical engine that performs intrusion detection, (18)'s approach provides a solution to a NP-complete problem that grows exponentially as the number of intrusions grows. In this study we considered 24 intrusions, so the search space was $2^{24}$, i.e., it was on the order of sixteen million. If we consider one more intrusion, the search space is $2^{25}$, that corresponds to thirty two million in order of magnitude. This makes the GA engine an appealing tool in the search for intrusions in audit trail files. However, the fitness function proposed by (18)—with the parameters used in this paper—gives us high false positive and false negative rates, besides the trouble of setting those parameters. We propose a simple and effective fitness function that overcomes the false positive and false negative rate problem and that avoids the use of parameters.

Our question already arises: are there other heuristic methods that can accomplish this task better than the GA proposed? The only way to know it would be to try some and compare—as a matter of example we tried to use *neural networks* (NN) to solve the problem, and when the neural net converged we ran into the problem of the boundary that separates intrusions from no intrusions, because the solution vector converged to *real values* as is common in NN.

One ideal topic to consider is the standardization of the audit trail file. Such a standard would have the principal benefit that it would enable logs generated by different operating systems to be reconciled. Organizations have multiple computers from different branches. This can result in multiple operating systems and applications. With the audit trail standardized, the analysis of logs by a central engine is simpler, because that engine need not to know the types of operating systems that generated those logs (25).

The field is deep and there are promising new ways to think about it. There are new paradigms to explore and we can use computers themselves as the vehicle, approaches such as immune systems and neural networks have been developed in order to improve this mechanism.

## REFERENCES

[1] R. G. Bace, *Intrusion Detection*. USA: MacMillan Technical Publishing, 2000.

[2] B. C. Tjaden, *Fundamentals of Secure Computer Systems*. Franklin and Beedle & Associates, 2004.

[3] ISC, "Internet Systems Consortium. ISC internet domain survey," 2006, accessed Feb. 2006. [Online]. Available: www.isc.org

[4] CERT, "CERT Coordination Center and Carnegie Mellon University. CERT/CC statistics 1988-2003," 2006, accessed Feb. 2006. [Online]. Available: www.cert.org/stats/cert_stats.html

[5] B. Schneier, *Secrets & Lies: digital security in a networked world*. Wiley Computer Publishing, 2000.

[6] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[7] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[8] K. J. R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[9] ——, *Genetic Programming*. Encyclopedia of Computer Science and Technology, 1997.

[10] K. John, *Genetic Programming*. MIT Press, 1992.

[11] ——, *Genetic Programming II*. MIT Press, 1994.

[12] e. a. Wolfgang Banzhaf, *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications*. The Morgan Kaufmann Series in Artificial Intelligence, 1998.

[13] J. Holland, *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[14] D. E. Denning, "An intrusion-detection model," in *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, 1986, pp. 118–131.

[15] M. Crosbie and E. H. Spafford, "Evolving event-driven programs," in *Genetic Programming*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. CA: MIT Press. Stanford University, 1996, pp. 273–278.

[16] M. Crosbie and G. Spafford, "Defending a computer system using autonomous agents," 1994, technical report No. 95-022.

[17] ——, "Applying genetic programming to intrusion detection," in *Papers from the 1995 AAAI Fall Symposium*, 1995, pp. 1–8.

[18] L. Mé, "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, 1998.

[19] P. Diaz-Gomez and D. Hougen, "Analysis and mathematical justification of a fitness function used in an intrusion detection system," in *In Proceedings of the Seventh Annual Genetic and Evolutionary Computation Conference*, 2005.

[20] ——, "Analysis of an off_line intrusion detec-

tion system. a case study in multi-objective genetic algorithms," in *In Proceedings of the Florida Artificial Intelligence Research Society Conference*, 2005.

[21] ——, "Further analysis of an off-line intrusion detection system: An expanded case study in multi-objective genetic algorithms," in *SCISS'05 The South Central Information Security Symposium*, 2005.

[22] ——, "Improved off-line intrusion detection using a genetic algorithm," in *In Proceedings of the Seventh International Conference on Enterprise Information Systems*, 2005.

[23] D. Fried and M. Zissman, "Intrusion detection evaluation," Lincoln Laboratory, MIT, Tech. Rep., 1998, http://www.ll.mit.edu/IST/ideval/, accessed March 2004.

[24] J. P. Anderson, "Computer security threat monitoring and surveillance," James P. Anderson, Co., Fort Washington, PA, Tech. Rep. 79F296400, 1980.

[25] M. Bishop, "A standard audit format," in *In Proceedings of the 18th National Information Systems Security Conference*, Baltimore, Maryland, USA, 1995, pp. 136–145.