From: AAAI Technical Report FS-95-01. Compilation copyright © 1995, AAAI (www.aaai.org). All rights reserved.

Applying Genetic Programming to Intrusion Detection

Mark Crosbie, Eugene H. Spafford
{mcrosbie,spaf}@cs.purdue.edu
COAST Laboratory
Department of Computer Sciences
Purdue University
West Lafayette
IN 47907–1398

Abstract

This paper presents a potential solution to the intrusion detection problem in computer security. It uses a combination of work in the fields of Artificial Life and computer security. It shows how an intrusion detection system can be implemented using autonomous agents, and how these agents can be built using Genetic Programming. It also shows how Automatically Defined Functions (ADFs) can be used to evolve genetic programs that contain multiple data types and yet retain type-safety. Future work arising from this is also discussed.

Introduction

Genetic Programming (GP) has been used to solve many problems that occur in the real world. Koza's book (Koza 1992) has numerous examples of using GP techniques to solve problems in a variety of fields. This paper presents a new application of genetic programming to solve a problem in the field of computer security. This application will exploit the learning power of GP. In the course of designing a solution to the problem, a novel use for Automatically Defined Functions (ADFs) (Koza 1994) was also discovered.

Background and Motivation

To understand the security problem being solved, this section provides a brief introduction and motivation. A computer system is secure if (Garfinkel and Spafford 1991)

> it can be depended upon to behave as it is expected to.

This is an intuitive definition — we generally will have confidence in a system if it behaves according to our expectations. More formally, security is often described in terms of *confidentiality*, *integrity* and *availability* (Russell and Gangemi Sr. 1991). Confidentiality is the property that private or secure information will not be accessed by unauthorized users, integrity is the property that the system and its data will remain intact and unviolated throughout the system lifetime, and availability is the property that the system will be available for use in the face of attempted attacks.

The security properties of a system are usually defined by the *security policy.* A policy is (Longley and Shain 1987)

> the set of laws, rules, and practices that regulate how an organization manages, protects and distributes sensitive information.

A security policy is necessary before any security system can be implemented or put in place on a system. Any potential attempt to attack a system can only be evaluated relative to the policy — if the policy prohibits some action, then executing that action is potentially an attack.

A security officer needs tools to enforce a level of security on a system. There are two main types of security tools on a system:

1. **Pro-active** — actively enforce the security policy of the system for every action executed by users. If the attempted action violates the policy, prohibit it from occurring.
2. **Reactive** — monitor the actions of users after they are complete and see if they violated the security policy. If a violation occurred, report this to a system operator.

In this context, a detection and reporting system would be a reactive tool. It would not actively stop an attack on a system, but would be able to detect it after the fact and report it to a system security officer. In our case, the detection system is called an *Intrusion Detection System.*

Intrusions and Intrusion Detection

An *intrusion* can be defined as (Heady et al. 1990)

> any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource.

Sometimes, these are also defined as instances of *misuse.* An intrusion is an attempt to subvert the security policy in force on a system to gain access to information, to alter the system's operation, or to deny the system to other users. Intrusions are enacted by users — a user runs a program which executes actions in excess of that user's authorization, for example. This would be classified as an intrusion. Similarly, a user who attempted to overload a system and prevent other users from working would also have initiated misuse (a denial of service). Note that this must be interpreted relative to a security policy — on some systems, this might be allowed (and expected) behavior.

To detect and report intrusions, a system must have some form of *intrusion detection system* (IDS) installed. An IDS must (Mukherjee, Heberline and Levitt 1994)

> identify, preferably in real time, unauthorized use, misuse, and abuse of computer systems.

As this is a reactive form of defense, it will not stop an intrusion from occurring. But without an IDS in place, the only evidence that an intrusion has occurred may be the disappearance of valuable files, or when the system halts. A human operator cannot reasonably be expected to filter megabytes of audit logs daily to detect potential intruders.

Components in an IDS

An *anomaly* is a deviation from an expected norm — some significant change in system behavior that may indicate a potential intrusion. Not all anomalies result from intrusions, so it is necessary to have some classification system to separate anomalous actions from intrusive actions. Typically this is done using an expert-system approach. A large ruleset specifies the security policy — this rule set forms a *model* with which the running system can be compared. System profile metrics are continually matched against this rule base, and if an intrusion is detected, an alert is raised. Examples of this are the *IDES* system (Lunt 1992) and *Wisdom & Sense* (Vaccaro and Liepins 1989). Both these systems derive metrics about the system's activity and then use an expert-system to decide if the calculated values indicate an intrusion.

To detect anomalies, some abstraction of system operation must be computed. Generally, this takes the form of a set of *metrics* which are computed from system parameters. These metrics can be as simple as average CPU utilization, or as complex as a command profile per user. By analyzing the metrics, it is possible to detect deviations from normal system behavior — anomalies. A metric is (Denning 1987)

> a random variable x representing a quantitative measure accumulated over a period.

A metric will encapsulate some pertinent information about an important system parameter that will be useful in detecting any anomalous behavior. Metrics are often computed by analyzing audit trail information: determining number of login attempts per unit time, for example.

An anomaly may be a symptom of a possible intrusion. Given a set of metrics which can define normal system usage, we assume that (Denning 1987)

> exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage.

This is the key idea behind an anomaly intrusion detector.

Finally, an IDS needs some method for comparing the computed metrics to the model and deciding if a sufficient variation exists to flag an intrusion. This may be based on statistical differences (e.g., between observed means), or it may use more complex covariance analysis (as in IDES (Lunt and Javitz 1992)).

Typically these mechanisms are built into one large module that performs all the actions of an IDS: it has the model encoded in it, it computes metrics from the audit data and it compares metrics with the model. The designers of the system have prespecified the nature and content of the model, they have decided on which metrics to compute and have established a mechanism to compare the metrics to the model.

If a new intrusion scenario arises, or if an existing scenario changes, modifying the IDS is cumbersome and complex. Rewriting rules in an expert-system is not a trivial task. Ideally, the rule base could be tailored to every end-user configuration that the IDS would potentially run on. This is impractical however — not all IDS users will be experts in expert-systems.

A Finer-grained Approach

Instead of one large, monolithic IDS module, we propose a finer-grained approach — a group of free-running processes that can act independently of each other and the system. These are termed *Autonomous Agents.*

An *agent* is defined as (Maes 1993)

> a system that tries to fulfill a set of goals in a complex, dynamic environment.

In our context, an agent would try to detect anomalous intrusions in a computer system under continually changing conditions: the agent would be the IDS. If an IDS can be divided into multiple functional entities that can operate in their own right, each of them can be an agent. This gives multiple intrusion detection systems running simultaneously — multiple *autonomous agents.* This approach is outlined in (Crosbie 1995). Figure 1 shows the architecture of the agent based solution, with sample agents monitoring I/O, NFS activity and TCP network connections.

The agents run in parallel in the system; in this design they are placed in the kernel. Audit data is fed from the kernel auditing system into the lower router. Its purpose is to separate the audit data into multiple *audit classes* and feed the audit records to the agents. Each agent then performs computations based on the contents of the audit records. An agent reports a *suspicion value* which indicates whether the agent considers the system (as viewed through the audit log records)

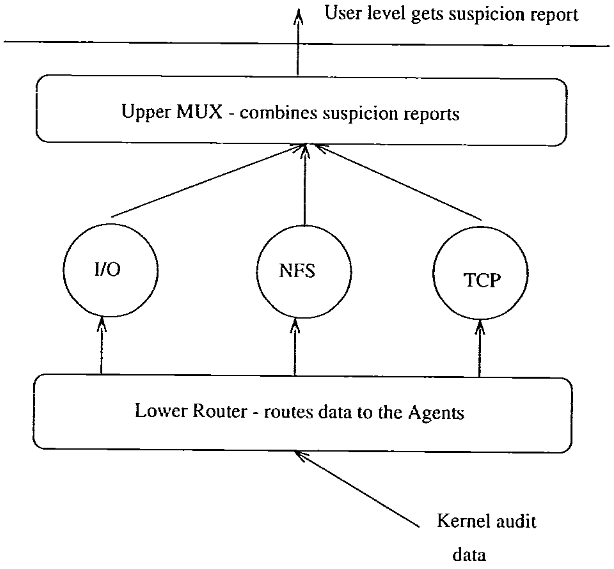
User level gets suspicion report
Upper MUX - combines suspicion reports
I/O
NFS
TCP
Lower Router - routes data to the Agents
Kernel audit data

Figure 1: Overall Agent-based IDS architecture

under the threat of an intrusion. Each agent reports its suspicion value to the upper multiplexor (Upper MUX) which then combines these individual values into an overall suspicion report, which is passed up to the user (the system security officer). The upper MUX can compute a simple average of the suspicion values, or more likely, a weighted average — if some agents are monitoring essential system parameters, their reports could be given more weight in an overall intrusion report.

Each agent is a lightweight program — it observes only one small aspect of the overall system. A single agent alone cannot form an effective intrusion detection system because its view of the overall system is too limited in scope. However, if many agents all operate on a system, then a more complicated IDS can be built.

Agents are independent of each other. They can be added to and removed from the system *dynamically*. There is no need to rebuild the whole IDS to add new agents. If an agent is created to monitor the network, it can be added to the IDS and placed into the kernel without affecting other agents already in place.

There are a number of advantages to having many small agents as against a single monolithic IDS. Having many agents provides a security system that has greater flexibility to add new monitoring capabilities, and to dynamically change current monitoring. It helps to reduce the single point of failure/single point of attack of traditional IDS. And, if properly done, the agents may be customized to each system, thus providing better coverage and reducing the effectiveness of clandestine attacks crafted on different systems with similar IDS.

A clear analogy can be drawn between the human immune system and this design. The immune system consists of many white blood cells dispersed throughout the body. They must attack anything which they consider to be alien before it poses a threat to the body. Sometimes it takes more than just one white cell to actually destroy the attacker. By having a large number of cells, the body is always able to defend itself in the most efficient way possible. If an infection occurs in one area, then cells will move to that area so as to fight it.

Using Genetic Programming for learning

One approach to building an IDS is to precode many possible intrusion scenarios into the system before it is installed. Thus, it comes with a static "knowledge-base" that it then uses to detect anomalies and intrusions. This has the shortcoming that a rule base has to be built by a group of experts and put in place on a system. Further, once it is there, it is difficult and expensive to maintain and update.

Our view of an IDS requires that the system be able to change over time to accommodate new information, and new threat patterns. That is, we want to be able to "evolve" the current agents, and train new ones. This will allow the system operator to customize the system and counter new threats. We can do this by providing a mechanism to "train" agents.

We selected Genetic Programming (GP) (Koza 1992) to implement this. This paradigm was chosen because it allows programs to be evolved and then executed in production environments. This was the overall goal of our investigation — to see if a collection of agents could be evolved that could be then placed in a system and left to monitor audit data. Genetic Programming was amenable to this goal — the final results of an evolution run is a set of programs. These programs are placed in a real system and are run continually to detect intrusions.

It is rare that programs evolved using GP are written in standard computing languages (such as C or PASCAL). Instead, a simple meta-language is often used which is tailored to solving a specific problem. Our solution is no different — the genetic programs are evolved in a simple language that has primitives to access audit data fields and manipulate them. In a stand-alone solution, these genetic programs are interpreted by an evaluator that supplies them with audit information. This is all contained within an agent in the final system. Figure 2 shows the internal architecture of an agent. The agent code is obtained from the evolution runs and is placed in the stand-alone agent to be interpreted by the evaluator. The evaluator obtains audit information from the *System Abstraction Layer* (SAL).

The SAL computes various statistics about the audit records and supplies them to the agents. It decodes the audit records and extracts the necessary fields. This abstracts the agents away from the format of the underlying audit data — agents can be developed independent of the audit record representation. The SAL provides a consistent set of primitives (e.g., average CPU utilization, average number of login attempts, etc.) to the agents across all systems.

Learning by feedback

Our learning model uses feedback — we present a scenario to the agents and evaluate them based on their performance. This is a separate program from the actual intrusion detection system. The training is batched and the best agents are then placed into the stand-alone IDS for use.

The scenarios were developed to present a wide range of potential intrusions to the agents. As important, however, is to train the agents not to flag legitimate behavior as intrusive. Thus the training scenarios are a mixture of both intrusive and non-intrusive activities. Each scenario has a probability associated with it that gives the likelihood that the scenario is an intrusion. This is used to assign a *fitness score* to each potential agent. This fitness score is then used by the genetic programming package to guide the evolution of the agents' code.

The scenario data is generated by a PERL (Wall and Schwartz 1992) script that executes commands at specified intervals. These commands correspond to what a potential intruder would execute. For example, one scenario attempts rapid connections to reserved ports (mail port, HTTPd port etc.), and has a high probability (90%) of being an intrusion.

What do agents monitor?

For our initial prototype, we decided to monitor network connections. We hoped to prove the agent concept viable (including the genetic programming component) by showing that agents could detect simple intrusions. Inter-packet timing metrics were computed by calculating the difference in time between two audit records that related to either a socket `connect()` or `accept()` call. The average, minimum and maximum inter-connection times were then updated. The destination port of a connect, and the source port of an accept were also stored. This process was repeated for every audit record that arrived. These metrics were made available to the agents via the System Abstraction Layer.

The metrics computed in this prototype were:

1. Total number of socket connections.
2. Average time between socket connections.
3. Minimum time between socket connections.
4. Maximum time between socket connections.
5. Destination port to which the socket connection is intended.
6. Source port from which the connection originated.

As each audit record has a timestamp, the first four metrics could be easily computed by calculating inter-audit record timing. The last two metrics were obtained by extracting the relevant information from the audit record.

Some potential intrusions that we hoped to detect with these metrics were:

- Rapid network connections to a specific port — *port flooding.*
- Searching through the port space for vulnerable services — *port-walking.*
- Gaining information from services (finger, httpd, NIS, DNS) — *probing.*
- Rapid remote login attempts — *password cracking.*

As this list shows, even with such simple metrics as inter-connection timings, we can detect a wide range of potential intrusions or anomalies.

GP building blocks

In Figure 2 the evaluator is shown querying the SAL for metric values. But how does it know which metrics the agent needs to access? The agent code is composed of a set of operators (arithmetic, logical and conditional) and a set of primitives that obtain the value of metrics. As is usual with Genetic Programming, these sets can be combined in any way to generate parse trees for solution programs.

These programs were then evaluated against a set of training scenarios (as outlined above) that allowed each potential agent to be assigned a *fitness score.* This

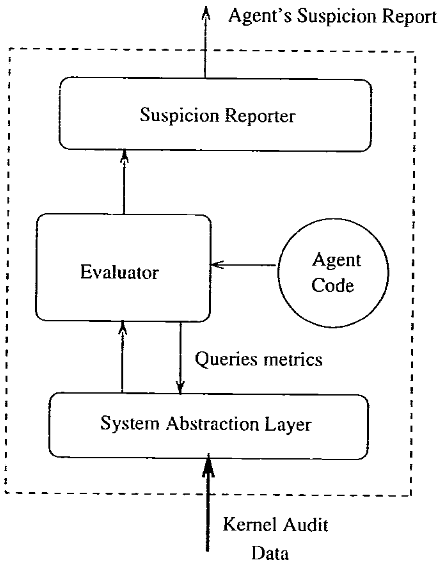

Figure 2: Internal agent architecture

score is based on how well the agent classifies the training scenarios. If an agent is presented with a training scenario that results in an intrusion, but it does not report a high suspicion after being fed the audit data from that scenario, the agent will be given a low fitness score. The fitness score is computed as follows:

The absolute difference between the agent's reported suspicion and the scenario's outcome is computed:

$$\delta = \mid outcome - suspicion \mid$$

A penalty is then computed based on how this scenario is ranked. Some intrusive behavior is more difficult to detect, in large part because it may be similar to allowed behavior. Thus, we weight our penalty based on the difficulty of detecting the intrusion. For example, if a certain scenario is an obvious intrusion, then agents which misclassify it should be penalized heavily. However, if a scenario is difficult to detect, the ranking is less severe and misclassification does not result in such a heavy penalty:

$$penalty = (\frac{\delta \times ranking}{100})$$

Finally, the fitness is computed based on the difference between the agent's suspicion and the actual suspicion:

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

A fitness of 0 indicates that the agent is predicting exactly the same outcome as the scenario — it is correctly classifying all the scenarios. A higher fitness score means the agent has deviated from the predetermined suspicion value associated with this scenario.

To actually report their suspicion, agents use two primitives to increase or decrease their suspicion values. Each agent has an internal variable giving its current suspicion, and this is reported to the upper MUX (in Figure 1).

Multiple types in a parse tree

The parse trees for the agents had to accommodate multiple data types. Audit data is a rich source of information and it contains many different types of data. Timing information will be of one type, whereas socket addresses or port numbers will be another. In our prototype, the following data types were used:

1. `time` — the interconnection timing information is in this form. It is a `long int` in our implementation, but this could change across implementations.
2. `port` — a port number. This is an `unsigned int` in our implementation, as port addresses range from 0 to 65535 in IP.
3. `boolean` — this is returned by some of the functions available to the agents.
4. `suspicion` — this is the current suspicion level of an agent. It ranges between 0 (not suspicious) to 100 (suspicious).

During the course of evolving an agent, it is likely that the parse tree will contain operations that attempt to combine operands of different types: adding a `time` type to a `boolean` for example. This leads to problems which will be discussed in the next section.

Problems with multiple types

In Koza's original work (Koza 1992), the parse trees for his genetic programs were constructed from uni-typed functions. Every function could manipulate the result of any other function. However, in real world situations, there is a need to handle multiple data types within a genetic program and still ensure *type safety*. This ensures that operators are passed operands that conform to a certain type-lattice. For example, adding a `boolean` type to an `integer` type would be prohibited.

Previous work has been done in the area of *strongly typed* genetic programming (Montana 1993). This addressed how to evolve parse trees that conformed to certain typing rules. However, despite the attraction of using this approach, we felt that it would involve significant re-writing of our GP package[1]. As our primary goal was intrusion detection, we chose the path of least resistance and implemented a different solution.

A solution using ADFs

Koza introduced the idea of an Automatically Defined Function (ADF) in his second book (Koza 1994). An ADF is essentially a subroutine that is evolved separately from the parent parse tree. It is this separate evolution that allows the generation of type-safe parse trees.

In our solution, each ADF performs a specific function — one tree evolves subroutines to monitor network connection timing, another evolves routines to monitor port addresses. Each of these trees will have a different set of primitives, and a different data type being manipulated. By keeping the evolution of the ADFs separated, there is no danger that unsafe parse trees may be generated. Any combination of any primitive within an ADF will be type-safe. This is a simple solution that avoids having to enforce strict typing rules during evolution. The type-safety is built in by design.

The ADF for a routine to monitor network timing has the following primitives available:

- `avg_interconn_time` — the average time between network connections.
- `min_interconn_time` — minimum observed time between two successive network connections.
- `max_interconn_time` — maximum observed time between two successive network connections.

These functions will return `time` type values. These values can be compared and manipulated with standard arithmetic operators (e.g. `+ - * / < > == !=`). A time value ranges from zero to some maximum time value which is implementation dependent.

The ADF for the port monitor routine has the following primitives available:

- `src_port` — the source port of the last network connection.

Table 1: Training scenarios

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%

- `dest_port` — destination port of the last network connection.

These primitive manipulate a `port` type, which is essentially an integer between 0 and 65535. Again, standard arithmetic and comparison operators are available.

To monitor connections to privileged ports, an ADF is used with the following two primitives:

- `is_priv_dest_port` — is the destination port of the last connection a privileged port (i.e. is it between 0 and 1023, inclusive)?
- `is_priv_src_port` — is the source port of the last connection a privileged port?

These functions both return a `boolean` value. This data type is manipulated using boolean operators `AND`, `OR` and `NOT`. Also available are some comparison operators `==` `!=` and the conditional `if` statement.

Each of these ADFs is evolved separately from the other — functional crossover never occurs between two different ADFs. Thus, the type-safety is preserved by the evolutionary process. However, this comes at the expense of having to maintain multiple populations per ADF — instead of having a population of individual parse trees, there are now four populations of parse trees; one per ADF plus a root tree. This is shown in Figure 3.

The root parse tree only has the ability to call the lower ADFs (similar to calling subroutines in conventional programming). Each ADF deals with a specific aspect of the audit data trail. ADF 1 handles the timing information about port connections, ADF 2 handles source and destination ports of connection and ADF 3 handles connection to privileged ports.

Results and Conclusions

To test the agents, scenarios were developed which we hoped would encode some potentially intrusive behavior related to network connections. The scenarios were artificial, given the limited scope of the metrics being computed. They were structured to train agents to detect obvious anomaly patterns — repeated connection attempts, connections to privileged ports and port walking. The training scenarios are shown in table 1.

[1] We were using `lilgp` v1.0 to evolve the agents.

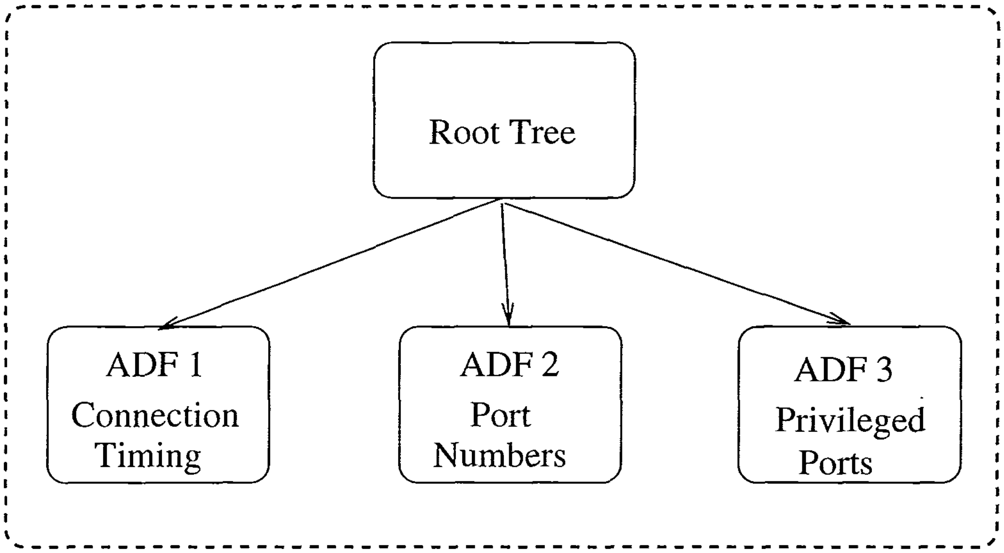

Figure 3: Each agent has multiple ADFs

The outcome of each scenario indicates the probability that that scenario results in an intrusion. The outcomes were decided *a priori*. This is an area for future investigation — developing good training scenarios involves addressing the issues of *coverage* and *depth*. We must ensure that the training scenarios will cover a wide variety of potential intrusions. This will ensure that the agents are able to handle as many potential intrusions as possible. Within each class of intrusion (i.e. network based, user based etc), we must ensure each agent has sufficient depth — that it has been exercised extensively within that class of intrusion. The above table of scenarios cannot hope to address these two issues, but it does reveal what is necessary to develop good training scenarios. Once a good collection of training scenarios are developed, they can be re-used many times to train new agents — the effort expended in developing good training scenarios is not wasted.

Three training runs were initiated with the above scenarios. Each run had slightly different parameters guiding the evolution. Once the evolution runs had completed. the best agent from each run was selected and placed in a test scaffold. This allowed audit data to be supplied to the agent without the surrounding evolution code. The idea behind the test scaffold is to gain understanding in what is necessary to run agents in a stand-alone environment. Once agents are evolved in the training system, they must be stripped down so that they can be placed in a production system and impose a minimum overhead.

The stand-alone agents were placed on a system running HP-UX 10.0. This ran the kernel is *trusted mode* which allows audit records to be generated when a system call is executed. The system calls to be audited could be configured using the standard HP-UX `sam` tool. To gather metrics on socket connections, the `accept()` and `connect()` calls were audited. Thus the audit trail would contain minimum extraneous records. This audit trail could then be fed to the System Abstraction Layer of each agent which would compute the metrics.

To test the agents with "real" data, auditing was temporarily enabled, and some commands were executed. These commands mimicked the actions of a potential intruder. The actions of a legitimate user were also simulated. Three test files were then supplied to the three best agents, and each agent reported a suspicion value. These values are summarized in table 2.

As can be seen, agents 2 and 3 performed better than agent 1. Agent 1 misclassified the last test suite as intrusive (73% probability). This test suite was not considered intrusive. Agent 3 classified this test suite with the least probability of being intrusive (25%). This is more in line with what we had in mind when executing these commands — they corresponded to normal system usage, and were not intrusive.

Agent 3 correctly classified the first test file as intrusive, but returns a 0% probability of intrusion for the second test case. This indicates that agent 3 is not yet perfect, but forms a good start point for continued evolution of solutions.

The first steps in improving this prototype would be to increase the variety of the training scenarios and develop more advanced attack scenarios with which to test the agents.

Table 2: Test cases and the agents' reports

Activities	Agent 1	Agent 2	Agent 3
Connections to privileged ports	83	100	98
Login then long pause, then logins	31	26	0
Logins and ftp with long pauses	73	47	25

Conclusions

It is impossible to compare a prototype of a new intrusion detector to a full-fledged system such as NIDES, but our results are encouraging. The prototype demonstrates that an intrusion detection system can be viewed as multiple functional entities that can be encapsulated as autonomous agents. Furthermore, we have demonstrated that Genetic Programming can be used as a learning paradigm to train our autonomous agents to detect potentially intrusive behaviors. We are confident that continued development will allow us to build more complex agents that will capture a wider variety of intrusive behaviors.

Future Work

This prototype development work has raised many questions, chief among which is how to make statements about the effectiveness of our intrusion detector. How can we be sure it will detect a specific intrusion? Can we compute a probability *a priori* of its effectiveness? What sort of overhead would such a system impose on a production system?

These questions will be investigated in future research.

References

Koza, John, 1992. *Genetic Programming.* MIT Press.

Koza, John, 1994. *Genetic Programming II.* MIT Press.

Denning, Dorothy E, 1987. An Intrusion Detection Model. *IEEE Transactions on Software Engineering* February: 222

Garfinkel S., and Spafford E. 1991. *Practical UNIX Security.* O'Reilly and Associates Inc.

Russell D., and Gangemi Sr. G. 1991. *Computer Security Basics.* O'Reilly and Associates Inc.

Longley D., and Shain M. 1987 *Data and Computer Security: Dictionary of Standards, Concepts and Terms.* Stockton Press.

Heady R., Luger G., Maccabe A., and Servilla M. 1990. The Architecture of a Network-level Intrusion Detection System, Technical Report, CS90-20. Dept. of Computer Science, University of New Mexico, Albuquerque, NM 87131.

Mukherjee B., Heberline L. T., and Levitt K. 1994. Network Intrusion Detection. *IEEE Network* May/June: 26

Lunt T, and Javitz H. 1992. A real-time intrusion detection expert system (IDES), Technical Report SRI Project 6784, SRI International.

Vaccaro H. S., and Liepins G. E. May 1989. Detection of Anomalous Computer Activity. In Proceedings of 1989 Symposium on Research in Security and Privacy.

Maes P. 1993 Modeling adaptive autonomous agents. *Artificial Life* 1(1/2).

Wall L., and Schwartz R. 1992 *Programming PERL.* O'Reilly and Associates Inc.

Montana D., May 1993. Strongly Typed Genetic Programming, Technical Report, Bolt Beranek and Newman Inc.

Crosbie M. and E. Spafford, October 1995. Defending a Computer System using Autonomous Agents. In Proceedings of the 18th NISSC Conference, October 1995.