Active Defense of a Computer System using Autonomous Agents

Technical Report No. 95-008

Mark Crosbie, Gene Spafford
COAST Group
Dept. of Computer Sciences
Purdue University
West Lafayette IN 47907-1398
{mcrosbie,spaf}@cs.purdue.edu

February 15, 1995

Abstract

This report presents a prototype architecture for an active defense mechanism for computer systems. The intrusion detection problem is introduced and some of the key aspects of any solution are explained. Previous attempts to use similar techniques are discussed, and their shortcomings are explained. A new architecture is proposed which uses Genetic Programming to evolve programs to detect anomalous behaviour in a system. This architecture is developed and evaluated. A sample genetic program is used to discuss some of the design aspects of the agents. Cooperative monitoring of NFS requests shows how the approach can be generalised. The discussion details some issues to be addressed and future research directions.

1 Introduction

Because of increased network connectivity, computer systems are becoming increasingly vulnerable to attack. These attacks often exploit flaws in either the operating system or application programs. The general goal of such intrusions is to subvert the traditional security mechanisms on the systems and execute operations in excess of the intruder's authorisation. These operations could include reading protected or private data or simply doing malicious damage to the system or user files.

In a running system there will be a variety of user and system processes performing jobs on behalf of the users. These jobs will perform different actions to accomplish their tasks: e.g. opening a file, writing to memory, communicating with other processes etc. So a view of a computer system running a user workload could be that of a continuous stream of actions on objects. In this view, the problem we are trying to solve can be stated quite succinctly as follows:

> Allow certain actions on certain objects in certain contexts. Closely monitor all other actions and treat

them as suspicious behaviour.

A working definition of suspicious behaviour is as follows: every system will have normal usage patterns. These patterns occur on a system wide level (e.g. the type and mix of jobs being run) and on a user level (e.g. average job length, type of job run, normal usage hours). Given these definitions of system usage patterns, anything that falls outside these norms will be considered *suspicious*.

Users sometimes break out of their normal usage patterns. This often occurs when a user must perform a task which they would rarely do. At other times they actually begin performing new tasks because of an external policy decision. Examples of these are: a user who normally uses the system to read mail starts to compile very large programs, or where a user who normally does development work on the system is moved to another system, leaving the original machine as just a mail home. In both these cases the usage pattern of the user has changed on the system in question.

We are proposing an *Intrusion Detection System* that will alert system operators to possible suspicious activity that may constitute an intrusion. The detection system will run independently of the jobs already on the system, and will provide continual information to an operator regarding any suspicious activity on the system. It must gather enough evidence to consider an action as suspicious before alerting the operator. In other words, it must be able to differentiate actual intrusions from small changes in user behaviour.

A key requirement is flexibility. It should be possible to specify what actions are to be allowed and disallowed initially. The detection system should also be trainable to recognise what actions are common on the system and adjust its detection mechanisms accordingly. Thus the detection system can be "tuned" to perform optimally in a given system. As system profiles change over time, the detection system will change with them to allow the newer activities, and possibly disallow earlier actions. This is accomplished by having the detection system learn by observation, deciding which actions constitute normal system behaviour, and which can be considered suspicious.

2 Intrusions and Intrusion Detection

An intrusion can be defined as [1]:

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

Intrusions are hard to catch because there are so many ways in which they may take place. Intruders can exploit both known architectural weaknesses in systems and inside knowledge of the operating systems to bypass the normal authentication process. A fix (or patch) to a flaw in a system may introduce a new flaw, or expose an existing one - giving rise to another opportunity for attack. Similarly, because of human factors, a given system may not have all (or any!) patches applied to it. So the *vulnerability state* of a system is in a continual state of flux. A good intrusion detection system must be able to deal with this.

Despite the many forms of intrusion, they can be catagorised into two main classes:

- *Misuse* intrusions are well defined attacks on known weak points of a system. They can be spotted by watching for certain actions being performed on certain objects.
- *Anomaly* intrusions are harder to quantify. They are based on observations of normal system usage patterns, and detecting deviations from this norm. There

is an inherent uncertainty about this form of detection - it may flag legal behaviour as illegal, or worse still, it may allow illegal behaviour to proceed, considering it normal.

As misuse intrusions follow well-defined patterns they can be detected by doing pattern matching on audit-trail information. For example, an attempt to create a setuid file can be caught by examining log messages resulting from system calls. This can be done using a pattern matching approach such as in [6], which is discussed in the next section.

However, anomaly intrusions are harder to detect. There are no fixed patterns that can be monitored for and so a more "fuzzy" approach must be taken. Ideally we would like a system that combined human-like pattern matching capabilities with the vigilance of a computer program. Thus it would always be monitoring the system for potential intrusions, but would be able to ignore spurious false intrusions if they resulted from legitimate user actions. It would rely on heuristics to decide this — they could either be pre-specified (by a human operator) or learned by the system over time. However heuristics will not always guarantee perfect accuracy, so another goal is to minimise the probability of incorrect classification. This is discussed more in the section on *Desired Characteristics*.

3 Related Work

The use of Artificial Intelligence techniques to help catagorise behaviours is not new, and the following section details some previous work.

Most previous work has focused on either *Behaviour Classification* or *Data Reduction*. The first case is where an attempt is made to decide whether a given set of behaviours constitutes an intrusive action. The second case is where a large data set (typically megabytes of audit-log data) is analysed for patterns of actions so as to reduce the amount of data to be handled by a human user.

The paper *Artificial Intelligence and Intrusion Detection* [4] outlines some methods by which classic AI techniques can be applied to the problem. It concentrates on the data reduction and behaviour classification problems. It outlines approaches involving both rule-based systems (such as *expert systems*) and classification systems (*neural-networks* or *classifier systems*).

The limitations of these approaches is that they require a lot of initial training and there is high maintainance during their lifetimes. In an expert system, the initial rule-base must be generated by hand using the knowledge of a human expert in the field. This is a time-consuming business, and probably quite expensive too. There is a more serious shortcoming in this approach though — not all experts know every vulnerability in a system, and even if they do, they cannot keep up to date with every vulnerability discovered. More seriously, they cannot discover vulnerabilities by considering the interactions of the many existing flaws in a system.

If the system profile changes considerably this rule-base will have to be redesigned to reflect new possible intrusions. This is an error-prone task — the new rules may not fully cover the set of vulnerabilities in the system. On a more practical level, the effort required to do this may prevent system administrators from keeping their rule-bases current, thus they will be operating with out-dated information in their intrusion detection systems.

Another example of a rule-based approach to intrusion detection is the IDES system [5]. It has a rule database which stores knowledge about vulnerabilities in the system, security policies in force on the system and past intrusions. From the system current state it attempts to match a rule which will classify the state of the system - has it been compromised or is it intact? This suffers from the limitations outlined above.

However, it has one important difference from the static rule-based approach — it remembers past intrusions. It builds on past experience in attempting to monitor a system[1]. We consider this to be an important feature of any intrusion detection system as new intrusion attempts are often slight modifications of previous attempts. More fundamentally, it is similar to the way in which humans approach an unfamiliar situation — "have I seen something similar to this before?"

Solutions which take a different approach to the problem, such as that proposed by Heady, Luger et al. in their paper on a Network Level Intrusion Detector [1], suffer from a problem of scaling. In this approach they use a *Classifier System*[2] to determine the state of their network. They gather metrics about network packets and from this try to infer whether they can classify the state of their network. However, this has two limitations:

1. It scales very poorly to a situation where many machines are on a high-speed network (such as an ATM or FDDI backbone) as the sheer volume of data to be processed would swamp any system.
2. The information they use to determine the network state is limited to packet header data. We feel that this is too limited in scope to be useful on its own. There is no information processed about the nature of the actions beyond that that can be deduced from the header. For example, there is no way of distinguishing a legitimate connection to the mail port from that of a possible intruder.

A system by Kephart [7] takes a similar approach to this paper by using the human immune system as a model for developing a virus detection and eradication system. However, his approach is specifically aimed at viruses on PC computers. He does not address the issue of anomalous behaviour or how to decide if a machine is undergoing an intrusion. Unfortunately the more interesting aspects of his work on virus-host attachment are proprietary.

However his paper describes some issues that must be addressed by our system. These include:

- Intruder recognition — deciding if an action by a user is possibly an intrusion.
- Learning about intrusions — similar to the IDES system mentioned earlier, his system attempts to learn about intrusions and use that knowledge in future decisions. We propose a similar mechanism.
- Response to an intrusion — once an intrusion is detected, how is it dealt with.

Finally, an approach which comes closest to the flexibility needed in a system like this, but does not possess the learning capability, is the intrusion detection model based on pattern matching as proposed in [6]. They show how attacks can be classified as patterns which match against occurances in a system. These patterns can encode dependencies between system conditions (i.e. event x must happen and so must y) and also temporal conditions (i.e. event x must happen before event y while condition z is true). This is a powerful method of detecting intrusions, but it relies on the patterns being generated beforehand. If the patterns are incomplete then there may be holes in the system's defenses. Again, the patterns may have to be re-generated if the system's behaviour changes due to a policy or operational change.

[1] using the useful rule-of-thumb in computing: *past behaviour is likely to predict future behaviour.*

[2] A cross between an expert-system and a neural network. See Goldberg [2] for information.

4 Desired Characteristics of the Detector

From the above summary of some related work, certain key points emerge. A intrusion detection system should address the following issues, regardless of what mechanism it is based on:

- It must **run continually** without human supervision. The system must be reliable enough to allow it to run in the background of the system being observed. However, it should not be a "black box" — its internal workings should be examinable from outside.
- It must be **fault tolerant** in the sense that it must survive a system crash and not have to have its knowledge-base rebuilt at restart.
- On a similar note to above, it must **resist subversion**. The system can monitor itself to ensure that it has not been subverted.
- It must impose a **minimal overhead** on the system. A system that slows a computer to a crawl will simply not be used.
- It must **observe deviations** from normal behaviour.
- It will be **easily tailored** specifically to the system in question. Every system has a different usage pattern - the defense mechanism should adapt easily to these patterns.
- **Changing system behaviour** over time as new applications are added means it must cope with changes in the system profile over time.
- It must be **difficult to fool**.

The last point raises an issue about the type of errors likely to occur in the system. These can be neatly catagorised as either **false positive**, **false negative** or **subversion** errors. A false positive occurs when the system classifies an action as anomalous (a possible intrusion) when it is a legitimate action. A false negative occurs when an actual intrusive action has occurred but the system allows it to pass as non-intrusive behaviour. A subversion error occurs when an intruder modifies the operation of intrusion detector to force false negatives to occur.

False positive errors will lead users of the intrusion detector system to ignore its output, as it will classify legitimate actions as intrusions. The occurances of this type of error should be minimised (it may not be possible to completely eliminate them) so as to provide useful information to the operators. If too many false positives are generated, the operators will come to ignore the output of the system over time, which may lead to an actual intrusion being detected but ignored by the users.

A false negative error occurs when an action proceeds even though it is an intrusion. False negative errors are more serious than false positive errors because they give a misleading sense of security. By allowing all actions to proceed, a suspicious action will not be brought to the attention of the operator. The intrusion detection system is now a liability as the security of the system is less than it was before the intrusion detector was installed.

The subversion error is more complex and ties in with false negative errors. An intruder could use knowledge about the internals of an intrusion detection system to alter its operation, possibly allowing anomalous behaviour to proceed. The intruder could then violate the system's operational security constraints. This may be discovered by a human operator examining the logs from the intrusion detector, but it would appear that the intrusion detection system still *seems* to be working correctly.

Another form of subversion error is fooling the system over time. As the detection system is observing behaviour on the system over time, it may be possible to carry out operations each of which when taken individually pose no threat, but taken as an aggregate form a threat to system integrity. How would this happen? As mentioned previously, the detection system is continually updating its notion of normal system usage. As time goes by a change in system usage patterns is expected, and the detection system must cope with this. But if an intruder could perform actions over time which were just slightly outside of normal system usage, then it is possible that the actions could be accepted as legitimate whereas they really form part of an intrusion attempt. The detection system would have come to accept each of the individual actions as slightly suspicious, but not a threat to the system. What it would not realise is that the combination of these actions would form a serious threat to the system.

5 Prototype Solution

As seen above, typical detection systems take the form of a monolithic block of code which sits either in the system kernel or on top of it and monitors all requests passing into the kernel. In the proposed solution, this approach is abandoned in favour of a group of free-running processes which can act independently of each other and the system. These are termed *Autonomous Agents*[3]. They are trained to observe system behaviour and flag any behaviour that they consider to be anomalous.

In this prototype, the agents will monitor the network traffic on a system. They will interface to the network via the *DLPI* [9] module provided with SunOs 5.x. This does not mean that the agents will only work on Sun machines. They require that an interface be provided that allows them access to network traffic. They access this interface through certain well defined primitives. How the actual data is provided to them is irrelevant.

They will be trained to detect anomalous activity in this traffic by being subjected to a training phase by a human operator. The operator will present different styles of network traffic (both intrusive traffic and neutral traffic) and guide the learning of the agents. Note that the agents use Genetic Programming to actually learn, the operator does not have to explicitly adjust the operation of any of the agents. This is described in detail below.

5.1 Design Overview

Figure 1 gives an overall view of how the agents operate. At the lowest level is the raw network interface itself. In this prototype implementation, this is the Sun DLPI interface [9]. It provides an interface to allow programs to transmit and receive raw datalink-level frames. This system does not generate any new network data, so only the receive capabilities are used. The system can gather data from the network and encapsulate it in a form that can be presented to the agents.

Above this lies the Network Primitives layer. This takes the raw network data from the DLPI interface and encapsulates it in such a way so as to allow the agents to handle it. The agents will require the values of various fields in the network packet header, plus a variety of aggregate values, such as average packet size, inter-packet arrival times and time-of-day. These values must be either derived from the packet data or from outside system sources.

The agents operate above the Network Primitives layer. Each agent is actually a pro-

[3] See the article by Maes [11] for an introduction to autonomous agents.

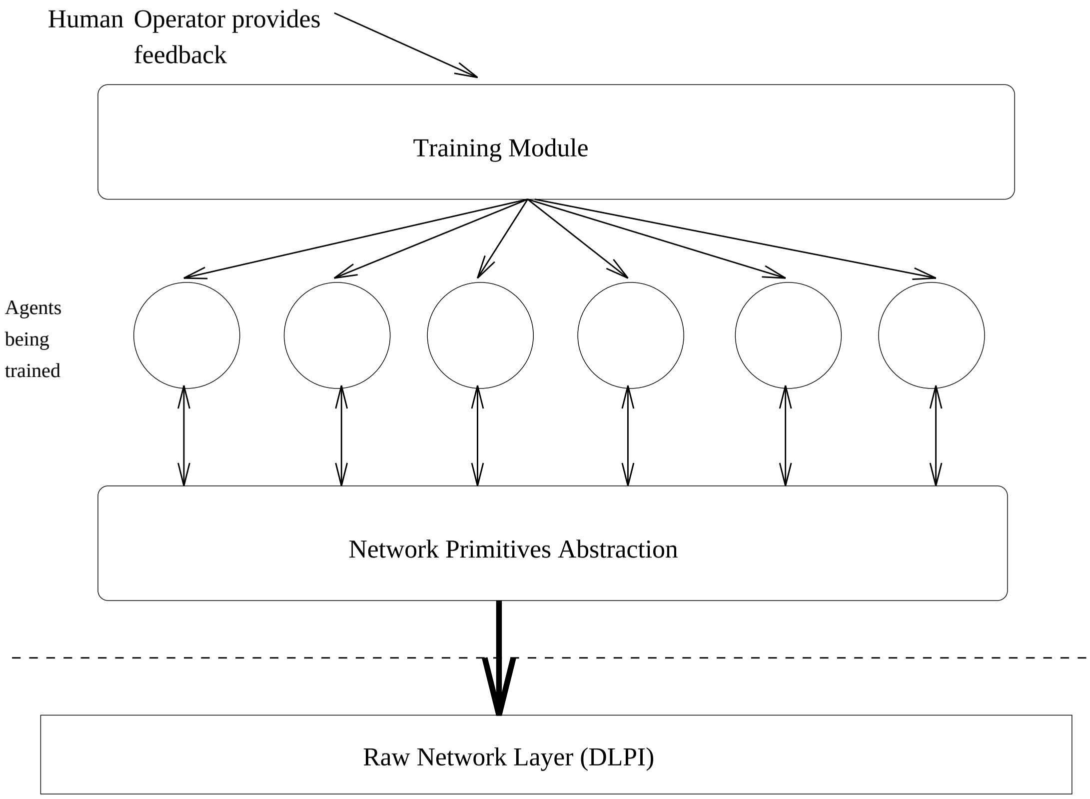

Figure 1: Architectural Overview of Agents in the system

gram which can be represented as a parse tree for a simple language. This language allows the agents to inspect the contents of network packets and perform operations based on this information. The network packet information is obtained from the underlying network primitives layer. The actual mechanisms of this, plus an example of an agent are described in the *Internal Design of the agents* section.

Above this lies the Training Module. Before the agents are allowed to monitor a system they must be trained to correctly respond to intrusions. They must also be trained to minimise the number of false positives (spurious intrusion reports) generated. This involves human interaction with the agents via this module. Once the agents have been trained they can be placed in a system without this module in place. The training is by a feedback mechanism — the operator provides an input describing whether the agents' actual behaviour was close to the desired behaviour for the given traffic pattern presented to them. It is similar to the training phase in neural networks.

6 Internal design of the Agents

We propose using the Genetic Programming [3] paradigm as a basis for the internal design of the agents. In this paradigm, populations of programs are evolved to solve a specific problem. The problem often has no singular correct solution, or the solution is very expensive to compute. The possible solution programs are represented as parse trees for a simple meta-language and these parse trees are manipulated by operations similar to those found in natural genetics. After time the population of programs converges on a particular program which gives the optimal solution to the problem.

Figure 2 shows a simple parse tree for an agent. This parse tree corresponds to the following block of pseudo-code:

```
for-each-packet do
  if( ip-destination-address-of-packet
           is-not-equal-to my-ip-address )
      then generate-a-suspicion-broadcast
  endif
endfor
```

The *Terminals* in the parse tree (the primitives `IP-DEST`, `MY-IP` and `RAISE`) obtain their values from the abstraction layer beneath the agents (see Figure 1). In this simple example, the primitive `IP-DEST` would obtain the IP Destination address for the current packet from the abstraction layer and then the `IP-NEQ` function would compare that address to the IP address of the system (given by the `MY-IP` primitive).

What this simple agent does is to raise the suspicion level (explained in the next section) of all the agents if it sees a packet that arrived at this machine, but had a different IP destination address from the one on this system. This may or may not be a useful thing to do, but it may perform some function in conjunction with the other agents on the system at the time.

6.1 Cooperation of multiple agents

One of the key ideas behind the Autonomous Agent approach is to evolve many agents at the same time. This allows greater scope for flexibility, with each agent monitoring a small aspect of the overall network traffic (as in the example above). However, the agents must cooperate together in order to detect intrusions. The above example probably does constitute suspicious behaviour (how would a packet arrive on this machine with an IP address different from our own, assuming only one network

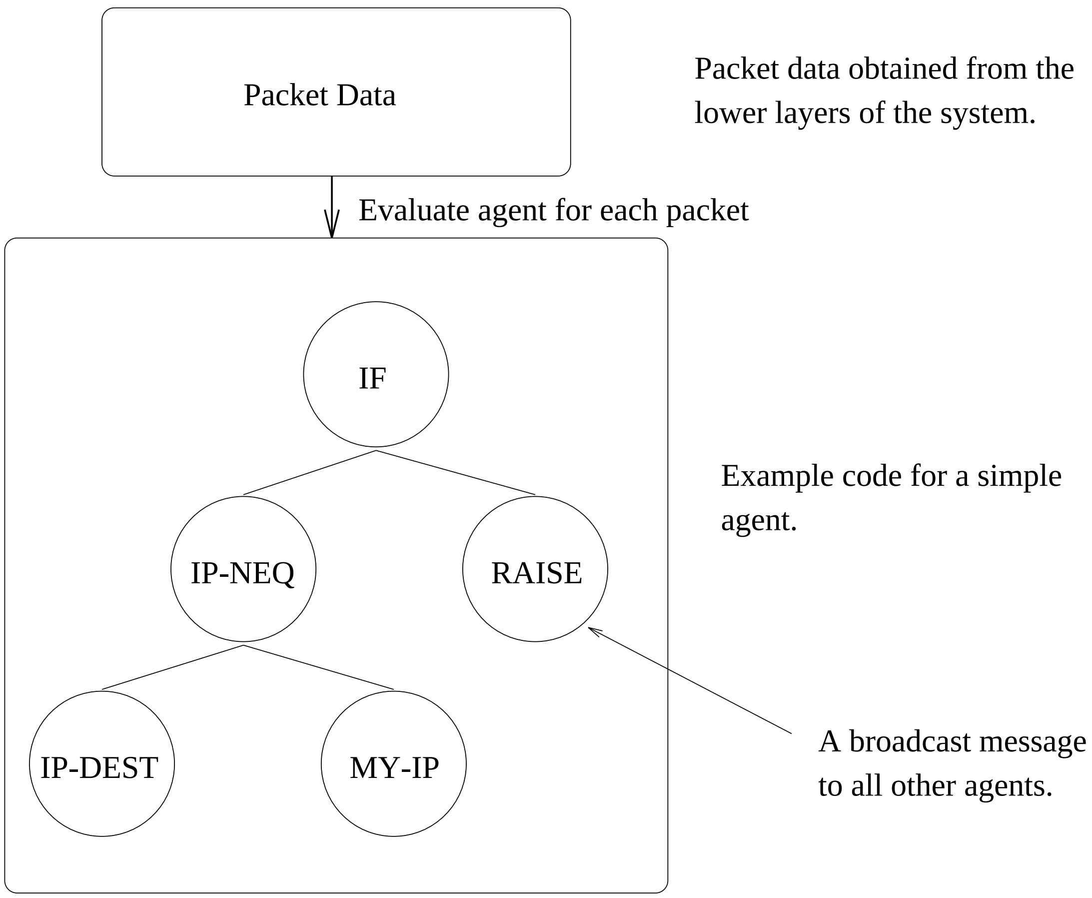

Figure 2: Sample internal parse tree for an agent

interface?). In other cases it would take a couple of agents together to cover all aspects of a possible intrusion (e.g. one agent monitors for UDP packets, another looks at destination ports in those packets and another monitors to see where packets are coming from). To be effective, these agents must be able to communicate their suspicion amongst themselves.

This is what the `RAISE` primitive does. It indicates that this agent believes that there is a possibly suspicious activity occurring and wishes to notify the other agents about this. As successive agents analyse the packet data, they too may make suspicion broadcasts. Eventually the general level of suspicion will rise above some pre-set threshold, and the system will indicate a possible intrusion to the operator.

7 Extending the approach

This section describes an example of how the approach described above can be applied in a more general case. There are three subsystems being monitored in the system — the network, the NFS device driver and the disk subsystem. The network connection has an agent which monitors the source address of incoming connections. If it sees one it has not seen before, it considers this as suspicious behaviour. Two agents are monitoring the NFS server. One of them analyses requests for NFS handles and another is monitoring all write requests. Finally an agent is monitoring the disk subsystem itself for writes to specific system directories. This is shown in Figure 3. Here an intruder is attempting to use a valid NFS handle to write to a system directory on the local disk. The intruder is coming in over the network from a previously unrecognised machine.

In this scenario, a write request comes in from a system X which agent A has never seen before. This causes the agent to become suspicious — it raises its suspicion level and sends a message out to other agents on the system. In this case the network connection is to the NFS server. This in itself is not enough to make agent A trigger an intrusion alert. However, agent B is monitoring requests for NFS handles and has received A's notification of suspicion. This, couples with its observation of the NFS request from X makes agent B increase its suspicion level, and broadcast this to the other agents.

Agents C and D have received these previous broadcasts and take them into account when they monitor actions. When the intruder at X issues a write request, agent C will have sufficient evidence to raise its suspicion level and broadcast this. Finally, when agent D sees a write to a system directory, its suspicion level has gone above a threshold value (due to all the earlier broadcasts from the other agents) and it will inform the operator of a possible intrusion.

This shows how the agents can cooperate to achieve the final goal of detecting an intrusion. Notice how each agent monitors for very common activities — agent B is monitoring NFS handle requests, a very common occurance in a networked environment running NFS. However, it is only when a sufficient weight of evidence is gathered by all the agents working together that an alarm is raised.

How do agents move from a suspicious state back to their normal state? The agents can let their suspicion level decrease over time. If an agent receives a suspicion broadcast, it will increase its suspicion level. If this is not followed up by any broadcasts from other agents, then it will move back into its normal operation state and continue monitoring.

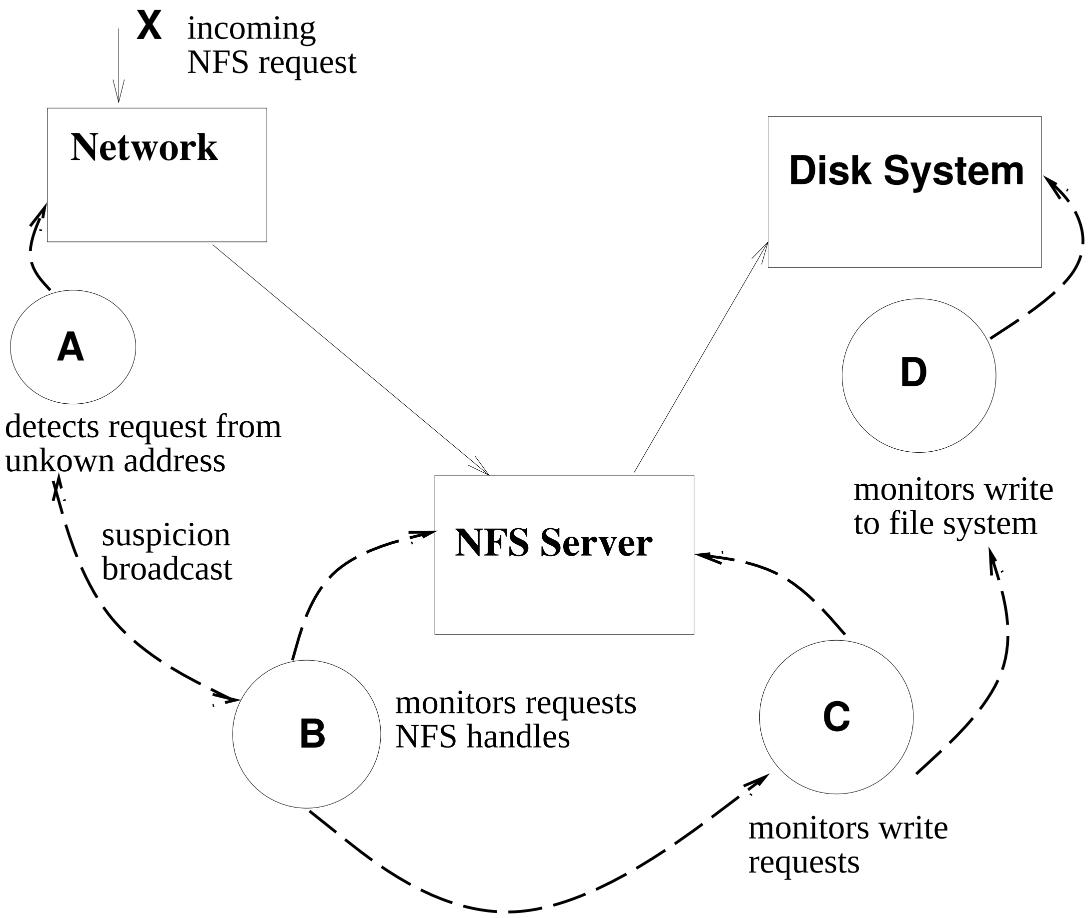

Figure 3: Active agents monitoring a NFS write request

8 Advantages and Disadvantages of the Autonomous Agents approach

The advantages of the proposed system are numerous. They are detailed below.

- **Easily Tailored**
  By having many small agents which observe system behaviour the detection system can be tailored to a system's needs in the most efficient way possible.

- **Trainability**
  The ability to be trained is an advantage in that the human operator can identify major threats to be monitored and teach the agents to recognise these threats above all others. Once the major threats have been identified, the agents are free to evolve mechanisms to monitor for other, less obvious threats.

- **Efficiency**
  Obviously, users do not want a degradation in the performance of their system. The individual agents must be optimised to perform their monitoring in the most unobtrusive way possible. The primitives used by the agents are very simple and can interface cleanly with an existing network-layer interface. Once the training phase is complete, the agents will impose a low overhead on the system.

- **Fault Tolerance**
  If the system they were monitoring were to fail, the agents would not lose any state. As they encode their behaviour internally as actual code, restarting the agents would leave them in exactly the same state as before. They can resume monitoring the system without any degradation in performance.

- **Graceful degradation**
  Similarly, if some agents are compromised, the system's defenses don't disappear. A graceful degradation in the system's ability to defend itself occurs - the best that can be expected in a case such as this.

- **Resilience to subversion**
  If a defense system is subverted by an attacker it is worse than useless - it gives a false sense of security. But knowledge of a particular agent on a system does not give knowledge of the operation of other agents - they all evolve under different conditions. Moving over to another system means that the agents there are slightly different so it is not a simple matter to subvert them. This is an important advantage.

- **Extendible**
  The agents could easily be modified to operate in networked environment where they actually migrated from system to system over the network. They could track anomalous behaviour over the network, and also move to systems where they would be most useful.

- **Scalability**
  The agents approach scales nicely to larger systems - simply add more agents and increase their diversity. Taking the whole notion to a network level also leads to an interesting insight - network agents which migrate around large networks and monitor network traffic for suspicious behaviour.

Of these the most important are, we believe, the ease of tailoring agents to your system, the resilience to subversion exhibited by agents and the highly scalable nature of the agents approach.

There are some drawbacks to the autonomous agent approach. They impose an **overhead** on the system as they will consume both memory and CPU cycles in order to monitor for intrusions. This is a cost of any intrusion detection system however, and the cost must be weighed up against the benefits of having a protection mechanism in place. **Training** the agents to monitor the system takes time. Unlike a solution which aims to be generic for every system, the autonomous agents will be tailored specifically for the system being monitored. This means that time must be spent analysing what is to be monitored before the agents can be placed in the system. The possibility of **false positives** must be minimised so as to make the intrusion detector a useful security tool. As in any intrusion detection system, if the agents are **subverted** then the intrusion detector becomes a security liability. Because the agents are distributed throughout the system and monitor many different system parameters, they are more immune to this sort of attack.

We feel that these disadvantages are outweighed by the flexibility of our approach. As this is a new field of investigation we feel that there is much to be discovered by using this paradigm for intrusion detection.

```
[ Note to reviewers:
   Experimental data should be
   available by the final May
   submission date. This will
   be included in the final
   version of this paper. ]
```

9 Discussion

There are a number of advantages to having many small agents as against a single large one. A clear analogy can be drawn between the human immune system and this proposal. 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.

We believe that this approach will lead to a more efficient and flexible approach to intrusion detection. It also appeals to our intuition to look to Nature for guidance when faced with tough design obstacles. This system is in the spirit of Evolutionary Computing, and yet still applicable to the Computer Security field. We feel that this cross-field development is the key to the advantages of this solution.

There are some issues that must be addressed as part of this research. Choosing the various **primitives** necessary for the meta-language in each agent will determine how well they can monitor network traffic. If the primitives chosen are too low-level then the agents may take longer to evolve more meaningful detection mechanisms. However, if they are at too high a level the agents may miss out important data which could be used in detecting a possible intrusion. For example, the TCP sequence number spoofing attack could be missed if the sequence number field was not available to the agents [10].

How are the agents to detect the actual **behaviour** of other processes? Are they to run in a protected mode, or are they just normal user processes with extra privileges? Should they gather their information from the standard system logs or will they use extra information (from device drivers or kernel routines, for example).

As every computer system is different, the ability of the agents to be **trained** is a major advantage. How this is to be undertaken will be investigated, as the effectiveness of the training will influence the ability of the agents to protect the system. Currently a "learning-by-feedback" model is proposed where a human operator will evaluate the agents based on their ability to detect known intrusions.

The issues of **testing and maintenance** will be considered, along with other issues such as how the agents are to interface with their operator and what level of knowledge will the operator need to effectively operate the agents. It is desired that the agents can be trained and installed without requiring extensive knowledge of genetic programming or security. However, for this initial prototype some knowledge may be required.

Once a working prototype is built, feasibility studies will be conducted to see whether a full version of the system would be practical.

References

[1] R. Heady, G. Luger, A. Maccabe, M. Servilla. *The architecture of a network level intrusion detection system.* Technical Report, University of New Mexico, Department of Computer Science, August 1990.

[2] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addision-Wesley, 1989.

[3] John Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection.* MIT Press, 1992.

[4] Jeremy Frank. *Artificial Intelligence and Intrusion Detection: Current and Future Directions.* Division of Computer Science, University of California at Davis, CA 95616.

[5] T. Lunt, H. Javitz, A. Valdes et al. *A Real-time Intrusion-Detection Expert System (IDES)*, SRI International Technical Report, SRI Project 6784, February 28, 1992.

[6] Sandeep Kumar, Gene Spafford. *A Pattern Matching model for Misuse Intrusion Detection*, Proceedings of the 17th National Computer Security Conference, October 1994.

[7] Jeffrey O. Kephart. *A Biologically Inspired Immune System for Computers.* High Integrity Computing Laboratory, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. To appear in Artificial Life IV, R. Brooks and P. Maes, eds., MIT Press, 1994.

[8] W.Z. Venema, *TCP WRAPPER, network monitoring, access control and booby traps*, UNIX Security Symposium III Proceedings (Baltimore), September 1992.

[9] Neal Nuckolls, *How to use DLPI*, Internet Engineering, SUN Microsystems.

[10] Steven Bellovin *Security Problems in the TCP/IP Protocol Suite*, AT&T Bell Laboratories, Murray Hill, New Jersey 07974. Computer Communication Review, Vol. 19, April 1989.

[11] Pattie Maes *Modeling Adaptive Autonomous Agents*, Artificial Life, Vol 1 No. 1/2, Ed: Christopher Langton, MIT Press, 1993.