Mobile agents for intrusion detection in wireless ad-hoc networks

by

Ryan Dean Hammond

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Information Assurance

Program of Study Committee:
Johnny Wong, Major Professor
Steve Russell
Thomas Daniels

Iowa State University
Ames, Iowa
2004

Copyright © Ryan Dean Hammond, 2004. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master's thesis of

Ryan Dean Hammond

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

**LIST OF FIGURES** .................................................................................................................. v

**ABSTRACT** .............................................................................................................................. vi

**CHAPTER 1: Introduction** ........................................................................................................ 1

1.1 Motivation ............................................................................................................................... 1
1.2 Wireless ad-hoc networks ..................................................................................................... 2
1.3 Intrusion detection system ..................................................................................................... 4
1.4 Mobile agents ......................................................................................................................... 6
1.5 Contribution ........................................................................................................................... 8
1.6 Roadmap ................................................................................................................................ 9

**CHAPTER 2: Related Work** ..................................................................................................... 10

2.1 Mobile agents for IDS ......................................................................................................... 10
2.2 IDS in wireless ad-hoc networks ......................................................................................... 12
2.3 Event correlation ................................................................................................................. 13
2.4 Summary ............................................................................................................................... 14

**CHAPTER 3: System Design and Implementation** ................................................................. 15

3.1 System design ....................................................................................................................... 15
3.2 Algorithm development for mobile agents ........................................................................... 17
3.3 MAIDS system architecture ............................................................................................... 20
3.4 System implementation ....................................................................................................... 21
3.5 Implications .......................................................................................................................... 25
3.6 Summary ............................................................................................................................... 26

**CHAPTER 4: System Testing** .................................................................................................. 27

4.1 Attack scenarios ..................................................................................................................... 27
4.2 Implications ............................................................................................................................ 34
4.3 Summary ............................................................................................................................... 34

**CHAPTER 5: Conclusion** ....................................................................................................... 35

5.1 Summary ............................................................................................................................... 35
5.2 Future work ........................................................................................................................... 36
BIBLIOGRAPHY ................................................................................................................. 37
APPENDIX A: Related Work Chart ................................................................................... 40
APPENDIX B: Agent Code ................................................................................................. 41
APPENDIX C: Mobile Transition ....................................................................................... 50
LIST OF FIGURES

Figure 1.1  Wireless ad-hoc network .......................................................3
Figure 3.1  Software layer design ..............................................................15
Figure 3.2  Mobile agent algorithm .............................................................17
Figure 3.3  Correlation rules .................................................................19
Figure 3.4  Correlation table .................................................................19
Figure 3.5  Agent transition .................................................................20
Figure 3.6  MAIDS system architecture ...................................................20
Figure 3.7  Hardware and software of laptop and palmtop .........................24
Figure 4.1  Correlated events ...............................................................28
Figure 4.2  Nmap port scan .................................................................30
Figure 4.3  Attack environment .............................................................32
ABSTRACT

While wireless ad-hoc networks continue to become more popular, the number of attacks on computer systems increases each year. These attacks are common occurrences in both the wired and wireless network environments. The wireless ad-hoc environment is susceptible to many attacks. Wireless ad-hoc networks are vulnerable to common wireless attacks (e.g. jamming) and attacks more specific to the ad-hoc environment (e.g. sleep deprivation attacks).

Encryption and authentication mechanisms alone have never been enough to prevent intrusions. Encryption can usually be broken, even if a brute-force attack is required, and authentication can easily be defeated (e.g. stealing a password or gaining access to a host which is already authenticated). Because of this, a second line of defense is needed. Intrusion detection systems have proven to be effective at providing this second line of defense. Establishing this second line of defense in a wireless ad-hoc network though brings with it many challenges. Such challenges consist of dealing with the ease that hosts enter and leave the network at random, of hosts being physically attacked or stolen and the possibility of an attacker disrupting the network through data route changes. This thesis aims to research agent-based intrusion detection systems and provide the design and implementation of an intrusion detection system based on mobile agents in a wireless ad-hoc network.

Utilizing research from intrusion detection systems, mobile agents and event correlation, we design a system for detecting intrusions in wireless ad-hoc networks by means of mobile agents. With the design created, we implement our mobile agent intrusion detection system using mobile devices set up in a wireless ad-hoc network. The system is then tested against three attack scenarios—two real-time, online attacks and one off-line attack.
CHAPTER 1: Introduction

1.1 Motivation

Data communication networks have become a vital part of today’s society. These networks transmit phone calls, internet traffic and banking transactions among other things over long distances. As demand for access to these networks has exploded, a new breed of data network has become popular—wireless networks. These wireless networks have quickly become the cutting edge technology for schools, universities, businesses and airports.

Wireless networks introduced a new freedom to its users. Individuals can now access the Internet without plugging a cable into their computer; they can leave their previously stationary workstations, and now freely move around while continuing to access the network through a common access point.

As wireless networks quickly gained popularity, yet another breed of data network evolved. This new type of network is called a wireless ad-hoc network. These ad-hoc networks allow individuals to set up private peer-to-peer networks anywhere without the need for traditional network infrastructures.

While wireless ad-hoc networks continue to become more popular, the number of attacks on computer systems increases each year. These attacks are common occurrences in both the wired and wireless network environments. The wireless ad-hoc environment is susceptible to many attacks. Such networks are vulnerable to common wireless attacks (e.g. jamming) and attacks more specific to the ad-hoc environment (e.g. sleep deprivation attacks [7]).

Encryption and authentication mechanisms alone have never been enough to prevent intrusions. Encryption can usually be broken, even if a brute-force attack is required, and authentication can easily be defeated (e.g. stealing a password or gaining access to a host which is already authenticated). Because of this, a second line of defense is needed. Intrusion
Establishing this second line of defense in a wireless ad-hoc network brings with it many challenges. Such challenges consist of dealing with the ease that hosts enter and leave the network at random, of hosts being physically attacked or stolen and the possibility of an attacker disrupting the network through data route changes. This thesis aims to research agent-based intrusion detection systems and provides the design and implementation of an intrusion detection system based on mobile agents in a wireless ad-hoc network.

1.2 Wireless ad-hoc networks

Wireless data networks have become commonplace in such places as airports, schools, universities and businesses. As more and more devices become capable of utilizing these networks and computing becomes more ubiquitous, a new type of data network is emerging—the wireless ad-hoc network. A wireless ad-hoc network is any network with a dynamic topology and no fixed infrastructure where inter-device communication is performed without the use of wires. In this thesis, we will concentrate on the network based on the IEEE 802.11 protocol. Furthermore, by our definition, there is no special hardware required for the network to operate. In a typical environment, routers and/or switches are necessary to pass a message from one device to another. In the wireless ad-hoc network, this is not necessary. Routing in this type of network is performed by the same devices which are sending and receiving messages. Consider Figure 1.1. If device A wants to send a message to device D, A will first send the message to B, then B will send the message to C, and finally C will deliver the message to D.

Wireless ad-hoc networks allow end-user devices to communicate directly with one another, forming a communication network anywhere, at any time. A common example of such a network is a group of students wishing to share digital information with each other. For instance, student A has a laptop and students B, C, and D have PDAs. The four students can form an ad-hoc network with each of their devices and run peer-to-peer applications. Similarly, these networks could be utilized for city-wide network coverage, vehicle communications, sensor networks and space missions.
Security concerns are inherent to wireless ad-hoc networks. Because these networks communicate wirelessly and have weak encryption standards, they are susceptible to security issues such as lack of ability to contain signal propagation, denial of service attacks against the communication channel and cryptanalysis attacks. The ad-hoc property adds even further concerns such as trust amongst the hosts.

The ad-hoc environment uses trust relationships between participating devices. When a node A, wants to send a message to node D, that message may first have to pass through nodes B and C. Nodes A and D both trust B and C not to breech the confidentiality, integrity and availability of that message. So B and C are trusted not to tamper with the message, and also trusted to pass the message towards D instead of merely holding the message forever or dropping it.

Figure 1.1 Wireless ad-hoc network
1.3 Intrusion detection system

An intrusion detection system (IDS) employs various mechanisms for finding an "intrusion." This intrusion can be defined as any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource [13]. An example of an intrusion would be a virus getting onto a home computer and deleting user files. The virus intrusion would have compromised the availability of those user files.

Intrusion detection systems are categorized into two models: misuse detection and anomaly detection. The misuse detection model uses known signatures to identify an intrusion. A common use of this model is an antivirus application for a home computer. This antivirus application uses signature files to scan the local computer for matching viruses. When a new virus becomes known, a new signature file must be installed in order to detect the new virus. The problem with the misuse detection model is that it can not detect new attacks without a signature. This is not the case with the anomaly detection model.

The anomaly detection model uses a "trained" normal profile of the system, and identifies anything out of the ordinary. A downfall of the anomaly detection model is its susceptibility to high false alarm rates. This model is more likely to identify an intrusion when there really isn't an intrusion. The reason for these false positives is because the system is constantly changing and the anomaly detector is not "learning" these changes quickly enough. For instance, if the system is trained on the normal behavior of a particular user and that user changes his or her routine one day, the system may see this behavior as an anomaly. Everything the user is doing may be perfectly legitimate, but the system is not accustomed to this particular change in behavior.

Intrusion detection systems can collect data in one of two ways - network based or host based. Network based data collection means all network traffic which can be "seen" by the device is collected. The intrusion detection system analyzes all the traffic on a network segment promiscuously. As an example of network based data collection, imagine a person standing in a crowded room. Everyone in that room is participating in different
conversations. Although the room is very noisy, our subject is able to hear everyone's conversations and understand what is being said.

On the other hand, host based data collection only collects network data destined for that host. The intrusion detection system in this case analyzes audit logs, system processes and network traffic locally. Going back to our example of the noisy room, if our subject is practicing host based data collection, he or she can only hear the communication that another person is saying directly to him or her.

Intrusion detection systems are in no way perfect. Common shortcomings of these systems include [24]:

- **Lack of efficiency:**
  - IDSs often attempt to process events in real time. In large networks, this is difficult to do when using network-based data monitoring, and packets get dropped. Regardless of network size, host-based IDSs also tend to slow down the system the IDS resides on.

- **High number of false positives:**
  - Accurately detecting attacks is very difficult. IDSs often report false alarms because of their imperfection and over-sensitivity to what constitutes an intrusion.

- **Burdensome maintenance:**
  - In misuse IDSs, signatures of intrusions are used to determine what is considered an intrusion and what is not. Periodically, these signatures need to be updated, which may require specific knowledge of how the system works to accurately update the system. Likewise, in anomaly detection the system must be periodically trained on normal system behavior which could also require specific knowledge of the system.

- **Limited flexibility:**
  - IDSs have proved to be difficult to move from one environment to another. This is especially true with a misuse IDS which has been trained for a
particular environment. Additionally, many IDSs must be restarted to effect any changes.

- Vulnerability to direct attack:
  - Most IDSs utilize a hierarchical structure for the different components. This results in a single point of failure further up the hierarchy. An attacker needs only to target a node higher in the hierarchy to take down the branch of components under that node.

- Vulnerability to deception:
  - An attacker may use specially crafted packets to fool the IDS. The attacker can alter fragmentation, sequence numbers, and packet flags in order to fool the IDS.

- Limited response capability:
  - Most IDSs are more concerned with detecting an intrusion than on preventing the intrusion. Typically, an administrator is given the task of responding to the intrusion once detected by the IDS. This gives the attacker time to continue the attack until the administrator has time to respond.

Currently, using mobile agents in IDSs does not solve all these shortcomings, but attempts to improve on some of these weaknesses.

1.4 Mobile agents

A mobile agent is a self-contained program capable of functioning autonomously. This program is able to carry out activities in an intelligent manner and is responsive to changes in the environment [12]. More simply stated, a mobile agent is a program that moves from host to host and performs a task at each host. This task might be statically or dynamically chosen for each host. This means the mobile agent may do something different on device A than it does on device B. Since a mobile agent is autonomous, the agent decides on its itinerary and may suspend its execution at any time and continue at the next destination [9].
There are a number of advantages for using mobile agents in an intrusion detection system [24]:

- **Overcoming network latency:**
  - Mobile agents can carry out tasks at the remote host, allowing the agents to respond in real time to changes in the environment.

- **Reducing network load:**
  - Mobile agents are capable of performing code execution on the same host as the data being used for that computation. There is no need to move data across the network to the agent.

- **Autonomous and asynchronous execution:**
  - If a portion of the system is damaged or lost, the mobile agents can continue to function and can repair (e.g. by cloning themselves) the damaged portion if necessary.

- **Dynamic adaptation:**
  - Agents are capable of adapting to their environment. If an agent is in danger it can move to another host, clone itself in case it is destroyed, or ask other agents for assistance.

- **Platform independence:**
  - Agents can move from host to host regardless of the hardware or software on each host. Agents only require agent platforms and these are available in many languages compatible with numerous software platforms.

Additionally, certain properties of wireless networks also cater to mobile agents. One such property is the ease at which the agents can be added to the environment. Since a mobile agent is autonomous and does not rely on outside support, a new agent can be injected into a network as long as an agent server is available. As new hosts enter into the wireless network, minimal effort is needed to integrate those hosts into the system.

Another favorable property of wireless networks is the lack of persistent connections, as required by the agent. In a typical client/server paradigm, communication is performed by
one host sending a request and then waiting for a response from another host. This communication is performed while keeping a connection live between the two hosts. Since agents move themselves to different hosts, they need not keep any connection open between themselves and another host at all times.

Agent platforms have been built in a variety of programming languages. Recursion Software’s Voyager [20] and IBM’s Aglets [10] are two agent platforms that are developed using Java. Gypsy [4] is a platform written in Java and Python, and Agents for Remote Action (Ara) [23] is written in Tcl and C, with plans to port to Java.

1.5 Contribution

This thesis focuses on using mobile agents in wireless ad-hoc networks for intrusion detection systems. The contributions of this research include: extending MAIDS to the wireless ad-hoc environment, implementing mobile agents in the wireless ad-hoc environment to detect intrusions, and using the mobile agents to perform event correlation in the wireless ad-hoc environment.

Extension of MAIDS to wireless ad-hoc network

Mobile agents are capable of moving themselves autonomously from host to host and executing their code at each of these hosts. We extend the Multi-Agents Intrusion Detection System (MAIDS) [9] from the wired environment to the wireless ad-hoc environment and develop a mobile agent platform to function in the wireless ad-hoc network for detecting intrusions.

Implementation of mobile agents for wireless ad-hoc networks

Our system design and implementation demonstrate the viability of using mobile agents for intrusion detection in wireless ad-hoc networks. We show how our agent-based IDS can be used to alert system administrators of an intrusion through three attack scenarios in a wireless ad-hoc network and design agents for each scenario to detect the attacks.
Event correlation using mobile agents in wireless ad-hoc networks

Finally, our research uses mobile agents to perform event correlation in the wireless ad-hoc network. Intrusion detection systems often report many low-level alerts, where many of those alerts might be false positives, or too many of these low-level alerts may be ignored by the system administrator. To obtain a higher level alert and be able to see how that alert was generated, some type of correlation is needed. Our research demonstrates how mobile agents can be used to correlate events in a wireless ad-hoc network.

1.6 Roadmap

Our goal is to extend MAIDS to the wireless ad-hoc network, implement mobile agents in the wireless ad-hoc network and use the mobile agents to perform event correlation for detecting intrusions in the wireless ad-hoc network. Chapter 2 discusses related work in the areas of using mobile agents for intrusion detection systems, intrusion detection in wireless ad-hoc networks, and event correlation techniques. Chapter 3 presents our system design and implementation used in our research. Chapter 4 discusses the testing of our system through three attack scenarios implemented in this study. In Chapter 5, we conclude the thesis with discussions on future work.
CHAPTER 2: Related Work

Much work has been done in the areas of using mobile agents for intrusion detection systems, intrusion detection systems in wireless ad-hoc networks, and event correlation. An overview of those topics is presented here as they pertain to the scope of this research.

2.1 Mobile agents for IDS

There have been numerous approaches using mobile agents for intrusion detection [24]. While each of these approaches is different, most agree that a hierarchical structure is well-suited.

The authors of [13] discuss a framework through a hierarchical approach where multiple agents are stationed at each node. This can conceptually be visualized as a tree with leafs representing the agents. In this framework, the agents on each host pass messages to the local transceiver (machine) that further pass information to a monitoring station (root node). This approach allows the intrusion detection computation to be performed at any node in the tree where enough information is collected.

The intrusion detection agent (IDA) system [14] uses mobile agents to trace intruders through a variety of hosts. This system is composed of a manager component to analyze information gathered by the agents, sensor components on each host to monitor system logs for marks left by a suspected intruder (MLSI), tracing agent components which are dispatched by the manager component to trace the path of an intrusion until the agent finds the origin of the attack, information-gathering agent components which are activated by the tracing agent to return information related to the MLSIs back to the manager, and finally bulletin and message board components which serve as a common area to exchange information between the agents and between the agents and the manager. This system is set up in a hierarchical structure with the manager at the root and the agents at the leaves.
In a dynamic network, nodes are likely to come and go. Any statically-defined hierarchical approach is likely to fail when a node higher up the tier leaves the network. One solution to this problem is to become less reliant on the hierarchy. Using mobile agents in a fully decentralized manner places less emphasis on a hierarchical design and more on agent mobility [2]. The approach taken in [2] only requires each participating device to have an agent platform and sensor support in order for the intrusion detection system to function. The agent then roams the network looking for suspicious activity. Once that activity has been detected, the agent sends child agents to interrogate other hosts before returning to the parent agent to share the gathered information.

In order to reduce the amount of intrusion detection processing performed by any one node, [17] proposes a framework where different hosts may have different responsibilities while participating in the intrusion detection system. This approach still uses a hierarchical design with multiple agents per host, where the agents form a hierarchy in order to distribute the workload of the IDS. The difference is in how much information each host is accountable for processing. Chosen by an election process, certain nodes monitor the network traffic promiscuously, while other nodes monitor only local traffic. The idea behind this approach is to have a detection agent on each node monitor items such as CPU activity, I/O activity, and user operations for anything unusual [16]. The local agent can terminate the suspicious process if necessary. If an inconclusive event is detected, the agent gathers data from an elected nearby neighbor agent to determine the appropriate action.

A more robust framework was sought after in [3]. This scheme demonstrated how agents could operate independently and cooperatively while organizing an IDS with no single point of failure. This approach does not use a single repository for the data collected and the alert notifications, but rather the mobile agents use other mobile agents as data repositories. Since each agent contains knowledge about the detected events and each agent is capable of performing all IDS functions, an attacker must disable each agent in order to stop the IDS. If an attacker only destroys a few of the agents, the remaining agents continue to carry out the functions of the IDS and can eventually rebuild the destroyed agents.
Following in the hierarchical framework design and using both stationary and mobile agents, [9] outlines a multi-tiered design where mobile agents travel between systems gathering data from localized agents then correlate that information and return the result to the system administrator. This system is built upon a hierarchy of agents with different tasks. As data is gathered and cleansed by the lower level agents, the data is passed further up the hierarchy with the user interface at the top. The user interface allows alerts to be displayed to an administrator and also for new agents and hosts to be dynamically added to the IDS.

2.2 IDS in wireless ad-hoc networks

Mobile ad-hoc networks provide a unique environment for intrusion detection systems. This distinctiveness demands new intrusion detection methods to be found for such an environment.

One such proposed method involves a hybrid intrusion detection system composed from four different intrusion detection techniques [6]. These authors attempt to use statistical anomaly detection algorithms, mobile agents, peer-to-peer data sharing techniques from Indra [19], and data-mining algorithms to create a scalable framework for both wired and wireless networks. This is a very ambitious approach and the system has not been implemented.

According to Zhang and Lee [25], new techniques are needed to make intrusion detection more effective in the wireless environment. They suggest an IDS architecture be distributed and cooperative among all nodes and should use a statistical approach for anomaly detection. The key idea is to have each node monitor local trace data for intrusions at each networking layer. If a node detects anomalous activity, that node will begin a cooperative verification through the combination of neighbors' trace data. Zhang et al. implemented this approach in ad-hoc routing protocols using a wireless network simulator package, Network Simulator.
ns-2 [26]. Ripper and SVM Light were used to compute classifiers for use as anomaly
detectors. Simulations showed the detectors had good detection performance, with the best
performance from on-demand routing protocols.

2.3 Event correlation

An intrusion attempt discovered on one device may be a part of a larger attack. Event
correlation attempts to discover these larger attacks by chaining together related attacks in
hopes of finding a causal relationship of an attack that otherwise would not have been
discovered.

A detection scheme to relate events that occur at different hosts can be used to detect
distributed signatures and anomalies. Such a scheme was proposed that looks for patterns by
sending messages between sensors running on different nodes [1]. Pattern matching is then
used to trigger alerts while each node only makes local decisions based on those alerts. These
patterns are described in a pattern description language where a pattern definition consists of
an attack scenario name, a node identifier, and a pattern (list of events). When a set of events
match the criteria for a scenario, an alert is raised. This scheme functions without the use of a
central coordination unit, although a centralized server is used to push updated attack
signature databases to the remote sensors.

Mobile agents are also used to correlate events in dynamic networks [2]. The tracing of
chained telnet sessions is performed by an agent using helper agents that are deployed to
gather further data and then return to the parent agent to share results for correlation. In this
design, each device in the network only needs to have an agent platform available to
participate. One node runs the management console which keeps track of participating hosts
in the network, allows users to specify queries before launching an agent and also configures
remote sensors. When a query is made, an agent is sent from the management console to look
for certain patterns. Once the agent has finished carrying out its task, it returns to the
management console to report its findings.
Another model for event correlation uses pre-conditions and post-conditions to link events [18]. The pre-condition of an intrusion is the necessary condition for the intrusion to be successful. Even though it is a condition for the intrusion to be successful, it does not need to match an attack itself. The pre-condition could be as simple as the existence of a vulnerable service without the attacker ever actively discovering the vulnerability. The post-condition of an intrusion is a possible outcome state assuming the intrusion was successful. There can be many possible post-conditions for a single intrusion. For example, if an intrusion successfully installs back orifice on a system, two possible post-conditions are that system can be vulnerable to a denial of service, or can be used as a file server. As alerts are generated by an IDS, these alerts are compared against the rules in a database to determine if a correlation can be made. If a correlation is made, a hyper-alert is generated to represent the alerts involved. The rules for defining the pre-conditions, post-conditions and hyper-alerts must be previously defined and the system can only detect correlated alerts which match those rules. This framework was evaluated using the 2000 DARPA intrusion detection scenario datasets and showed a high percentage of correlating all related alerts in the datasets. One such correlation from that dataset consisted of an attack with five phases. Phase one was a Sadmind_Ping where the attacker found a vulnerable service. The second phase was a buffer overflow attack called Sadmind_Amslverify_Overflow. This was followed by Rsh alerts when the attacker installed daemon and master programs. The fourth phase involved alerts caused by the daemon and master programs communicating. Finally, the fifth phase was a distributed denial of service attack.

2.4 Summary

In this chapter, we have provided the necessary background on intrusion detection systems in wireless ad-hoc networks, event correlation and other work that is related to our research using mobile agents for intrusion detection. Based on the concepts introduced here, the system design and implementation used in our research will be presented in the next chapter.
CHAPTER 3: System Design and Implementation

This chapter details the system architecture and algorithm used in this research. Also included in this chapter is our implementation of the system design. Common ad-hoc devices include cellular telephones, laptop computers and personal digital assistants (PDAs). Other devices such as desktop computers can also be used to form an ad-hoc network. For the scope of this paper, all devices used in the ad-hoc network will be assumed to be mobile devices.

3.1 System design

The software employed in our research can be visualized as consisting of multiple layers, each layer building on the previous layer. Figure 3.1 illustrates these layers.

![Software layer design diagram](image)

Figure 3.1 Software layer design
- Mobile Agents for Detecting Intrusions:
  - Mobile agents make up the top layer of our design. This layer consists of single or multiple agents. These agents carry out any task (e.g. detecting intrusions) they are programmed to perform.

- Mobile Agent IDS:
  - The fourth layer manages the mobile agents and consists of the intrusion detection system. This layer can be anything from an administrator console to display agent messages, to a simple application to launch agents into a network.

- Mobile Agent Platform:
  - The third layer makes up the agent platform. Such platforms can be written in different programming languages such as Java (e.g. Recursion’s Voyager and IBM’s Aglets), Python (e.g. Gypsy) and Tcl (e.g. Ara).

- Java Virtual Machine:
  - The second layer consists of a Java virtual machine. This layer is needed when the higher layers are written in Java, and an interpreter is needed. This layer could be replaced with something other than a JVM if another programming language, such as Python or Tcl, is used in the Mobile Agent Platform layer.

- Operating System:
  - The bottom layer consists of the operating system. This layer can be any operating system such as Linux or Microsoft Windows since the next layer up consists of a virtual machine.
3.2 Algorithm development for mobile agents

Mobile agents make up the top layer of our system design. For each of the attack scenarios described in Chapter 4, a new agent is created to detect the attack. Figure 3.2 shows the algorithm used to create the software agent. This algorithm provided the basic structure of the agents and supplied a modular approach for adding new features to the agents.

1. Process Snort log for subset of alerts (as defined as rules in agent code)
2. Store those alerts which are related to intrusion
3. Look for correlations from stored events
   If (correlations found)
   Store those correlations
   Raise alert level
4. Move to next host

Figure 3.2 Mobile agent algorithm

The first thing the agent does when arriving at a new host is to read alerts from the Snort log file. This is performed in the agent code (see Appendix B) in the function `check4intrusions()`. Once an individual alert from the log is read in, the alert is compared against rules to determine if a hyper-alert can be created. These hyper-alerts store detailed information about the alert such as date, source IP, source port, destination IP, destination port, and time. Each alert read from the log file can generate a single hyper-alert if the alert name from the log file is the same as an alert defined in the rules. The hyper-alerts are stored in a table for future correlating.

The subset of alerts mentioned in Figure 3.2 is defined as rules in the agent code, as shown in Figure 3.3 [5]. These rules specify the pre-conditions and post-conditions for alerts. Figure 3.3 shows pseudo-code of how this is done in our agent that is used in our first test scenario. The alerts which are related to an intrusion are stored in a table as hyper-alerts. The hyper-
alerts are then correlated and stored in the table *CorrelatedAlert*. This table is set up as shown in Figure 3.4. The variable *id* stores the counter for keeping track of how many correlations are in the table. For example, the first alert inserted in the table would have an *id* of 1, the second an *id* of 2, etc. The variable *Cause* stores integer representation of the pre-condition for the correlated alert, and *Result* stores the integer representation of the post-condition for the alert. The *Events* variable stores the string representation of the correlated alert for displaying the alert to the administrator. An example of the *Events* variable would be 67286 → 67416 where these numerical values represent the Snort alert identifier.

Before the agent moves to another host, the new alerts in the correlation table are compared against other alerts in the table for matching values. This correlation is performed in the function *InsertCorrTable()* shown in the agent code presented in Appendix B. For example, if the table has an existing alert with the *Events* set to 67286 → 67416 and a new alert is ready to be inserted into the table with the new *Events* being 67416 → 67540, the values are compared. If a pre-condition of either matches a post-condition of the other, the preexisting *Events* is modified to reflect this. Using our example values, the new *Events* will be appended to the old, resulting in 67286 → 67416 → 67540. The resulting *Events* pre-condition becomes 67286 and the post-condition becomes 67540. This newly created alert is then inserted into the end of the table, and further comparisons are done on the remaining alerts in the table. Once the end of the table has been reached, the new alert with *Events* 67416 → 67540 is finally inserted into the table.

After the local log files have been processed, as mentioned above, the agent is ready to move to another host. To move to another host, the agent chooses a new destination from a list of known hosts with available agent servers running. This list is assigned to the agent before the agent is released into the network, and the agent can choose from the list randomly or according to a certain itinerary. Once the agent has chosen a new destination, the agent server serializes the agent and the agent data. The server then transfers the serialized data to the destination agent server. The receiving agent server deserializes the agent and its data, and begins execution of a callback function. This transition from a host to the destination host is
initiated by the function call shown in Figure 3.5. Once the `moveTo` method has been called, Voyager will automatically perform this serializing, transfer of data, and deserializing, along with execution of the callback function. The function `newHost.dest()` returns the IP address of the destination host. "Mainloop" is the callback function used in Voyager's `moveTo` method. This "mainloop" function will begin executing once the agent arrives at the destination host. The agent code for this example is included in Appendix B, along with the callback function in Appendix C.

```
while (not end of log file)
{
    input Events from Snort log file;

    if (Event_Name matches rule)
    {
        create hyper-alert;
        store hyper-alert in ConseqSet and PrereqSet tables;
    }
}
create correlated alert;
store new alert in CorrelationTable;
move to new host;

public void MyRules()
{
    rule = new RuleSet[6];
    rule[0] = new RuleSet("Sadmind_Ping", "OSSolaris", 0, "VulnerableSadmind", 0);
    rule[1] = new RuleSet("Sadmind_Amslverify_Overflow", "VulnerableSadmind",
                           "OSSolaris", 1, "GainAccess", 0);
                           "SystemCompromised", 2);
                           "SystemCompromised", 2, "ReadyToLaunchDDOSAttack", -1);
    rule[4] = new RuleSet("Stream_DoS", "ReadyToLaunchDDOSAttack", -1, "DDOSAgainst", 0);
}
```

Figure 3.3 Correlation rules

```
class CorrelationTable implements Serializable
{
    int id;
    int Cause;
    int Result;
    String Events;
}
```

Figure 3.4 Correlation Table
3.3 MAIDS system architecture

Figure 3.6 shows the MAIDS system architecture. This architecture is composed of several layers with multiple agents at each layer.

Making up the bottom layer, stationary agents, known as data cleaning agents, gather information from system logs and audit data. These agents then arrange the information into a common format to be used by the next higher layer.
The middle layer of the MAIDS architecture is built from low-level agents. These mobile agents monitor and classify activities and events as they visit each of their associated data cleaning agents. The information at this layer is shared with other agents in the same layer and is also passed further up the hierarchy.

The top layer of the architecture in Figure 3.6 contains the user interface and data mining and fusion agents. These are the high-level agents. The data fusion and data mining agents collect data from lower level agents and mine knowledge from the database. The database stores information for use by the agents and for off-line training purposes.

The hierarchical architecture for this IDS agent system has the following advantages [9]:

1. The implementation of agents is efficient. Since low-level agents travel to monitored systems, mediator parts need not to travel. Many low-level agents are generated and migrating to monitored systems, so the mediator part does not need to be generated multiple times and does not need to migrate. This saves much network bandwidth and CPU time.

2. The layered system is easy to design and modify. A clear organization of the agents makes the system easy to maintain.

3. It provides platform independence. The lower levels that need to have contact with system logs are platform dependent. When a new operating system is added, only the lowest level agents (data collecting agents) need to be added.

### 3.4 System implementation

The system consists of numerous mobile devices as hosts. The hosts are all members of a wireless ad-hoc network. Each node in the system runs an agent server and includes the MAIDS software. Certain nodes also run Snort [22] as a local intrusion detection system. A more detailed look at the system is discussed in the following sections.
Hardware

The handheld devices in this research are the Sharp Zaurus SL-5500 [21] with a CompactFlash 802.11 Ethernet card. For the wireless card to work in ad-hoc mode, a new Linux kernel with wireless extensions enabled is required [11].

The other hardware is standard laptop computers running Microsoft Windows operating system. These computers have built-in 802.11 Ethernet cards.

With the PDAs acting like handheld Linux computers and communicating wirelessly with the laptops, the next step is to get the MAIDS software working on the devices.

Software

The software in our research is implemented in a layered approach (see Figure 3.1). The intrusion detection agents are implemented in the top layer. The next layer down is composed of the MAIDS software.

The MAIDS project is written entirely in Java making it compatible on many platforms. Employing lightweight agents, MAIDS is built upon a hierarchy of agents with different tasks. As data is gathered and cleansed by the lower level agents, the data is passed further up the hierarchy with the user interface at the top. The user interface allows alerts to be displayed to an administrator and also for new agents and hosts to be dynamically added to the IDS. The design of MAIDS is illustrated in Figure 3.3.

For agent mobility, MAIDS utilizes the Voyager Object Request Broker [20]. This agent platform supporting MAIDS consists of Voyager modules. This project takes advantage of many features from Voyager version 3.2. Both MAIDS and Voyager are written entirely in Java, so layer two of our design consists of Java Virtual Machine version 1.4.0 on the Windows laptops, and JVM 1.1 on the PDAs. Figure 3.5 summarizes the hardware and software that is used in this research.
Recursion's Voyager is chosen as the agent platform for our research because it has already been established as a working agent platform for MAIDS for the wired network in our previous project, is written entirely in Java, is a commonly used mobile agent platform, and has technical support available through Recursion Software, Inc. This platform provides the following useful functions for the agents in the IDS [9]:

- Mobility:
  - The Voyager ORB provides agent mobility from host to host. Any serializable object can easily be moved between Voyager servers. In the case of mobile agents, the agents continue executing after arriving at the new destination.

- Interfacing:
  - Voyager provides a means for dynamic aggregation of objects. This allows objects to be easily upgraded with new code and data at runtime, allowing the agents to remain lightweight as they traverse the network and then being updated with new services when necessary.

- Naming services:
  - The naming service allows access to many naming services (e.g. CORBA, RMI) through a single API. It also allows easy binding of objects.

Voyager is also written entirely in Java and thus supports a broader platform for simple code migration. Furthermore, Voyager makes available many modules that are beneficial for future research, including:

- Messaging:
  - Voyager provides an API for sending messages to an object regardless of the object type or location. This can be used to ease development of inter-agent communication within an IDS.

- Security:
This module provides a method for implementing SSL communication amongst objects. Additionally, agents can utilize this module to allow them to tunnel through firewalls.

• Transactions:
  o Using a two-phase commit, this module ensures distributed transactions are properly terminated. This could be used for research concerning critical decision making among agents in the IDS.

![Figure 3.7 Hardware and software of laptop and palmtop](image)

**Extending MAIDS**

A major problem encountered in extending MAIDS to the wireless ad-hoc environment had to do with agent mobility. The agent was unsuccessful, in the beginning of this research, at moving between physical machines. The agent would move between various servers hosted on the same machine (each server running on a different port), but the agent would not move when the destination server was on a different physical machine. After much trial and error in a wired network setup, it was discovered the MAIDS agents ran into problems when moving from host to host if the hosts were running different versions of Java Virtual Machines (JVMs).

The JVM compatibility problem was an important one. The PDAs run JVM version 1.1, and the other devices had various versions of the JVM installed. Ultimately, many versions of the
Java platform were tested including versions 1.3.1, 1.4.0 and 1.4.2. In our research, only version 1.4.0 was compatible with the JVM on the PDAs. Using this version of the JVM allowed the agents to move from host to host without any problems.

Finally, it was discovered that when compiling the MAIDS source code for the PDAs, the "-target 1.1" flag needed to be set as a compile option or else the code would not work properly. Once all the java problems had been overcome, there was still a problem with the Voyager servers.

In order for the agents to move from host to host, each destination host must be running a Voyager server. This server opens a system port for the agent to use during transitions, and also interprets the serialized agent data. A problem was discovered during this research between the Voyager server and the version of JVM on the PDAs. Voyager ORB versions 4.7, 4.6, 4.5 and 3.2 were tested on the PDAs. Version 3.2 was the only version that was successful in our research. The other versions of the ORB had a function that was making a call to the Vector.add() method, which is an invalid method in JVM 1.1.

With MAIDS functioning in the ad-hoc environment, the next phase of this project was to extend the application to detect attacks in this environment. The original design of MAIDS was tested with the FTP bounce attack [9]. To further add other attacks to which the MAIDS software system can detect, new agents and attack scenarios are needed.

### 3.5 Implications

In a mixed-architecture environment, there is a high probability of having software on certain hosts and not others. There is also a high probability of having similar software on multiple hosts, but with different versions of that software. Generally, software is assumed to be backward compatible with older versions. Our experience in this research has taught us that not all JVMs are compatible with one another. Having found software that is compatible with our system as a whole, we now have a working mobile agent IDS for our mixed architecture.
Whether the environment is wired, wireless or a mix of both, our system configuration will allow our mobile agents to visit the Linux-based PDAs and the Windows-based laptops.

3.6 Summary

This chapter presents the software and hardware configurations used, along with our system design and implementation. Also shown here are the MAIDS system architecture and the organization of the software layers. Using the agent algorithm and system implementation introduced here, Chapter 4 details the testing of the system through three attack scenarios that are implemented in this research and detected using mobile agents.
CHAPTER 4: System Testing

Using the system design and implementation described in Chapter 3, this chapter illustrates three attack scenarios and testing that is done for this research. These attack scenarios were implemented—two real-time, online attacks and one off-line attack, and are below.

4.1 Attack scenarios

Scenario 1: Stream Denial of Service (DOS)

The first attack scenario that was run in the wireless ad-hoc environment is an off-line attack. This scenario is aimed at making sure the agent can gather data at multiple hosts and then correlate that data to find intrusions. The technique for this off-line attack detection is adapted from [18]. This technique uses pre-conditions and post-conditions to link events. As alerts are generated by an IDS, these alerts are compared against rules in a database to determine if a correlation can be made. If a correlation is made, a hyper-alert is generated to represent the alerts involved. The test data is taken from that in [18], and the data file is divided into multiple pieces. The data file is divided by extracting arbitrary lines of data from the original file and copying these lines into new files. This is done in such a way that each new file is similar in length, and no two of these new files has data from the same line of data in the original file. The new files are then placed on different hosts in the network. An agent is created to visit each node in the wireless ad-hoc network, collect the alert data from these newly created files, and attempt to link events using pre-conditions and post-conditions. The rules used for determining whether a correlation exists or not are programmed into the agent code instead of using a database for the agent to interact with. As this is an off-line attack, the agent is programmed to report all correlated events back to the console after it has gathered the data from all hosts. The console application from MAIDS is then modified to display the results correlated by the agent. Figure 4.1 shows an example output of the correlated results on the console. The top line shows the event identifier and the bottom line shows the name of the event.
Executing the attack:
Since this is an off-line attack, the attack has already been carried out. The attack the agent will be monitoring is a denial of service attack. The original data for this scenario is from the 2000 DARPA intrusion detection scenario dataset LLDOS 1.0. This dataset contains multiple attacks in which an attacker probes hosts, breaks into those hosts, installs software, and finally launches a distributed denial of service attack against a further target.

Detecting the attack:
The actual attack in this scenario has been previously detected by an IDS. That IDS in turn created the alerts in the original dataset. The goal of this scenario is not to detect the attack (that’s already been done by the IDS), but instead to correlated the alerts generated by the IDS. This will provide a high-level representation of the alerts to reveal their causal relationships.

Correlating the attack:
This agent correlates the events of this attack in a decentralized fashion. Using pre-condition and post-condition rules (stored in arrays in the agent code) to link events, the events collected by the agent on each host are compared against the rules to determine if a correlation can be made. If a correlation is made, a hyper-alert is generated to represent the alerts involved. For example, let’s assume event A is a pre-condition to event B, and event B is a pre-condition to event C. Let’s further assume a post-condition of event C is a compromised system. The agent visits Host1 and reads the events from the data file (placed there earlier in the scenario, but could theoretically be dynamically updating in real-time). The agent discovers events A and B have both occurred on this host, checks those two events against the rules it knows of, and determines a hyper-alert should be created to link these two
events together. The agent stores this hyper-alert in its memory, looks for further correlations on the current host in the same manner, and then moves to Host2. The agent compares the events on Host2 with all other events on Host2 against the rules, generating hyper-alerts as necessary. The agent then compares the events on Host2 and the hyper-alerts in memory against the rules, creating "higher" hyper-alerts if the rules find a match between events. After that has finished, the agent moves to Host3 to perform the same task. At Host3, the agent finds event C and creates a "higher" hyper-alert with A, B and C. After the agent has visited all hosts in this network, the agent is programmed to write all hyper-alerts to the console machine (Figure 4.1) for analysis by an administrator.

**Scenario 2: Nmap Scan**

The second attack is a distributed Nmap [8] scan. Nmap is a program that is capable of scanning large networks in order to determine which hosts are up and what services are available on those hosts. An Nmap scan can be considered as an attack because it is one of the likely first steps an attacker uses in carrying out an intrusion. This type of scan is considered an attack at numerous universities and goes against their security policies. As such, hosts suspected of executing these scans are often disconnected from the network.

In this scenario, the agent travels between three hosts. The attacker, on a fourth machine, performed a randomized port scan on the other two hosts. Figure 4.2 illustrates this scenario. The job of the agent is to detect what appeared to be completely random (and few in number) port activity on each host, and correlate the aggregated results to decide if an Nmap port scan is being carried out on the ad-hoc network.
Executing the attack:
The attacker executes an Nmap port scan on the network. In this case, the attacker is trying to do this undetected. The attacker uses a SYN scan so systems are not completing handshakes and connecting to one another. This type of a scan works by sending a SYN packet from the attacker and waiting for an ACK packet to be received. The attacker then sends back a RST packet to close the connection. If a RST packet is received from the target instead of the ACK packet, then the attacker knows this port is not active and can not be used in a later intrusion attempt. To further reduce the chance of being detected, the attacker can scan the target machines at random time intervals and using pseudo-random port numbers while also
randomizing the hosts. This could be done by simply using the \texttt{--scan\_delay} and \texttt{--randomize\_hosts} flags with Nmap. For this particular scenario, system ports twenty through one hundred fifty were scanned.

\textbf{Detecting the attack:}
Each host in the network is running an IDS. For this experiment, that IDS is Snort. Snort detects the SYN packets from the attacker and generates an alert for the targeted packet. The mobile agent in this scenario is continuously visiting each of the hosts, one-by-one. As the agent is visiting a host, the agent reads alerts generated by Snort. If multiple Snort alerts are found by the agent on a host with each alert coming from the same source (i.e. the attacker), this agent remembers the port numbers and source address from those alerts. The agent continues to visit other nodes.

\textbf{Correlating the attack:}
If the agent finds similar alert patterns on the hosts (A, B, C) with the same source address as had been seen before, the agent correlates the events. If enough of these particular events are correlated, the agent takes a predefined action. In the case of this scenario, once one hundred unique ports have been discovered coming from the same host, the agent raises its alert level and prints a message to the screen of each host it visits in order to alert users of the scan. For each additional one hundred unique port scans found, the agent will raise the alert level again until the alert state is at red. The levels, from less severe to most severe, are: \textit{Green, Low Yellow, Medium Yellow, High Yellow, Red}.

\textbf{Scenario 3: Distributed real-time attack}
The third and final attack demonstrated a distributed attack resulting in a system being compromised. Figure 4.3 shows the various hosts in this attack. Host $A_0$ is the attacking machine. $T_0$ is the target for the attack. $C_0$ and $C_1$ are nodes which have been compromised by $A_0$ and will be used to carry out the attack on $T_0$. 
Executing the attack:
The first step in this attack is for the attacker, A₀, to gain access to C₀ and C₁. The attacker then performs a port scan from C₀ on the target machine. Following the port scan, the attacker attempts to obtain information about the services running on the open ports of the target. Once the attacker has this information, she launches a more intrusive attack from C₁ against the target. The intrusive attack in this scenario is the Nachi [15] worm infecting the target. This worm first sends an ICMP ping to the victim machine to see if the victim is listening. If a reply is received, the worm attempts to propagate. This is done by opening a remote shell on the victim and instructing the victim to download the worm from the attacker via TFTP. Once the download is completed, the code begins executing on the victim. Once code execution begins, this worm attempts to download a Microsoft patch to prevent other programs similar to it from infecting the victim in the same way. The worm then begins more attempts to propagate. This worm is considered more of a nuisance than a malicious worm.
The primary downside of this worm is the large volumes of ICMP traffic sent over the network. Other variants of this worm are more damaging to computer systems and can be used to test our system.

**Detecting the attack:**
While the attack is being carried out, an agent is moving among a number of hosts, including the target host, $T_0$. Each host the agent visited is locally running the intrusion detection system, Snort. Similar to Scenario 2, the agent is reading the log files produced by Snort at each host. The agent also uses the same method as in Scenario 2 to detect the port scan. As the agent is moving from host to host, it is carrying with it a signature for the Nachi worm. This signature is a pattern in the log file that will be produced by Snort upon detecting traffic from the worm. When the agent sees this pattern in the log file, it will remember that data as it continues to travel the network.

**Correlating the attack:**
The agent in this scenario is looking for three things: a port scan, a machine trying to obtain banner information from a service, and the virus signature. When the agent detects a machine trying to obtain banner information from a local service of the current host, the agent checks the port of the service against previously scanned ports. If the port which the local service is running on is a port that had been previously scanned, the agent correlates those two events together. When the worm is detected in the Snort logs, the port targeted by the worm is also checked against previously scanned ports. If there is a match, the events are correlated. The worm port is also checked against previous banner information port attempts. If a match is found, those events are also correlated. Once the worm has been detected, the agent raises its alert level status to red and prints the alert, along with the correlated events, to the screen of each host. As each correlation is made between these particular events, the alert level status is raised by one level. The levels, from less severe to most severe, are: *Green*, *Low Yellow*, *High Yellow*, *Red*. 
4.2 Implications

These attack scenarios show, through implementation, mobile agents can be used to detect intrusions in a wireless ad-hoc environment. We also see that the agents can distribute the IDS to take place on each host in order to collect data from the whole network.

In the wireless ad-hoc environment, there is no data concentration point. Implementing an IDS on a host in the network will not guarantee any intrusion detection for other parts of that network. Using mobile agents allows us to take the IDS to each host in the network to collect data instead of needing a data concentration point for data collection. Much research has suggested IDSs be distributed for the wireless environment. Our tests show mobile agents can be used to accomplish this distribution and collect data network-wide.

Using the event correlation in the wireless ad-hoc network allows the IDS to discover network-wide events along with same-host events. The mobile agents compliment this task by being able to visit the host in the network and collect events. An unfavorable property of using the mobile agents for event correlation though is the more events an agent correlates, the more data that agent has to move through the network each time the agent visits a new host. We did not run into issues with this in our research, but it is feasible for the agent to have slow transition times between hosts as the amount of data collected increases.

4.3 Summary

This chapter shows three attack scenarios that are implemented in this research. We see the agent detect Nmap scans, a virus propagating and see the agent correlate events. The next chapter gives the summary of this research along with future work that can be done in the area of using mobile agents for intrusion detection.
CHAPTER 5: Conclusion

5.1 Summary

The objectives of this paper were to develop an agent-based intrusion detection system and provide an implementation for using mobile agents for intrusion detection in a wireless ad-hoc network.

We have provided copious examples of agent-based intrusion detection systems. These systems ranged from hierarchical approaches to flat schemes. They also differed in using multiple agents in the network to using a single agent.

With an understanding of what has already been done in the area of agents and intrusion detection, our research is able to design a mobile agent to detect multiple attacks in the wireless ad-hoc environment. Armed with the capability to detect intrusions, our agent is able to correlate events to give an administrator a higher level view of those attacks. Our implementation demonstrates the viability of using mobile agents for intrusion detection in wireless ad-hoc networks.

Through this research, it has been shown that mobile agents are well suited for assisting intrusion detection systems in the wireless ad-hoc environment. As the hosts in an ad-hoc network move and the environment changes, mobile agents are capable of moving also; repositioning themselves to gain an advantage. The agents are also able to move from host to host autonomously making it difficult for an attacker to target a specific agent. And because agents are autonomous and are capable of cloning themselves, they can continue to function and even rebuild the system when parts are either under attack or destroyed. Lastly, agents are capable of moving directly to a source of information for analysis instead of having to move the data to the agent. This is very beneficial given the limited network bandwidth in wireless ad-hoc networks.
5.2 Future work

The area of intrusion detection has been studied for quite some time, and has even recently been studied in wireless networks. Looking at these systems from an ad-hoc environment poses new challenges and raises new areas of research.

One such area of future research is how to manage agents in an ad-hoc network. One agent alone can not be expected to monitor hundreds of hosts in a real-time system environment. Research needs to be done on how to dynamically create and allocate agents efficiently. Once the agent is created, there needs to be an effective method of assigning hosts for that agent so the agent can minimize the number of hops needed to visit the hosts. Likewise, a schema to allow efficient communication and data sharing amongst these agents needs to be studied.

Other work should focus on a framework of how to handle the different scenarios that may happen with hosts in the ad-hoc environment. Since the hosts are constantly moving, entering and leaving the network, how should the mobile agents respond? Also included in that framework should be criteria for handling the case where an agent is “carried off” the network by a host.

Finally, research should be conducted on creating a smart agent with machine-learning anomaly intrusion detection capability for the ad-hoc environment. This would allow a quick deployment of these agents into a previously unknown network.
BIBLIOGRAPHY


http://wwwagss.informatik.uni-ki.de/Projekte/Ara/status.html#interpreters, 1997.


## APPENDIX A: Related Work Chart

<table>
<thead>
<tr>
<th>Paper Title</th>
<th>Hierarchical Design</th>
<th>Mobile Agents</th>
<th># of agents in an N-node network</th>
<th>Event correlation</th>
<th>Implemented</th>
<th>Wireless Mobile Ad-hoc compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible, Mobile Agent Based Intrusion Detection for Dynamic Networks [2]</td>
<td>N</td>
<td>Y</td>
<td>&lt; N</td>
<td>Y</td>
<td>J, D</td>
<td>N</td>
</tr>
<tr>
<td>Intrusion Detection Techniques for Mobile Wireless Networks [26]</td>
<td>Y</td>
<td>Y</td>
<td>&gt; N</td>
<td>N</td>
<td>NS-2</td>
<td>Y</td>
</tr>
<tr>
<td>Decentralized Event Correlation for Intrusion Detection [1]</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
<td>J, D</td>
<td>N</td>
</tr>
<tr>
<td>Constructing attack scenarios through correlation of intrusion alerts [18]</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
<td>O, D</td>
<td>N</td>
</tr>
<tr>
<td>A Method of Tracing Intruders By Use of Mobile Agents [14]</td>
<td>N</td>
<td>Y</td>
<td>&lt; N</td>
<td>N</td>
<td>P, D</td>
<td>N</td>
</tr>
<tr>
<td>Our Approach</td>
<td>Y</td>
<td>Y</td>
<td>&lt; N</td>
<td>Y</td>
<td>J</td>
<td>Y</td>
</tr>
</tbody>
</table>

Y = yes  
N = no  
> N = greater than N agents required  
< N = less than N agents needed  
N/A = not applicable  
P = implemented using Perl  
D = implemented on desktops  
NS-2 = implemented on network-simulator2  
O = implemented offline  
J = implemented using Java
APPENDIX B: Agent Code

/*
MAIDS: Multi-Agent Intrusion Detection System
   Test Scenario 1 Agent
*/

package agent;
import MAIDS.*;
import java.util.*;
import java.lang.*;
import java.io.*;

class Table implements Serializable // Pre and Conseq tables
{    int ID;
     String En1;
     String En2;
     String time;
     String EventName;
}

class SmallTable implements Serializable //the correlated table
{    int id;
     int Cause;
     int Result;
     String Events;
}

class hostmachine implements Serializable
{    String name;
     int visits;
}

// Had to add this here for Serializability
class RuleSet implements Serializable {
    /**
    * PreArg and ConArg is the argument of prerequisite and consequence.
    * arg:-l-no argument;0-one argument and that is DestIPAddress;1-two
    * arguments and both are DestIPAddress; 2-two arguments and one is
    * DestIPAddress and the other is SrcIPAddress
    */
    String EventName ;
    String Prerequisite1;
    String Prerequisite2;
    int PreArg;
    String Consequence1;
    String Consequence2;
    int ConArg;

    public RuleSet(String e, String p ,int pa, String c, int ca){
        this.EventName=e;
        this.Prerequisite1=p;
        this.PreArg=pa;
        this.Consequence1=c;
        this.ConArg=ca;
    }

    public RuleSet(String e, String p1,String p2 ,int pa, String c, int ca){
        this.EventName=e;
        this.Prerequisite1=p1;
    }
public RuleSet(String e, String p, int pa, String c1, String c2, int ca) {
    this.EventName = e;
    this.Prerequisite1 = p;
    this.PreArg = pa;
    this.Consequence1 = c1;
    this.Consequence2 = c2;
    this.ConArg = ca;
}

public RuleSet(String e, String p1, String p2, int pa, String c1, String c2, int ca) {
    this.EventName = e;
    this.Prerequisite1 = p1;
    this.Prerequisite2 = p2;
    this.PreArg = pa;
    this.Consequence1 = c1;
    this.Consequence2 = c2;
    this.ConArg = ca;
}

public boolean IsRule(String e) {
    return (EventName.compareTo(e) == 0);
}

class HyperAlert extends Alert implements Serializable {
    String Prerequisite1;
    String Prerequisite2;
    String Consequence1;
    String Consequence2;

    public HyperAlert(Alert a) {
        super(a);
    }

    public void SetPrerequisite (RuleSet r[], int x) {
        if (r[x].PreArg == -1) /* no argument */
            Prerequisite1 = r[x].Prerequisite1;
        else if (r[x].PreArg == 0) /* one prerequisite argument and that is DestIPAddress */
            Prerequisite1 = r[x].Prerequisite1 + "(" + DestIPAddress + ")";
        else if (r[x].PreArg == 1) /* two prerequisite argument and both DestIPAddress */
            Prerequisite1 = r[x].Prerequisite1 + "(" + DestIPAddress + "")";
            Prerequisite2 = r[x].Prerequisite2 + "(" + DestIPAddress + "")";
        else /* two prerequisite argument and one is DestIPAddress and the other is SrcIPAddress */
Prerequisite1 = r[x].Prerequisite1 + "DestIPAddress " + "SourceIPAddress +";
Prerequisite2 = r[x].Prerequisite2 + "DestIPAddress " + "SourceIPAddress +";
}

public void SetConsequence(RuleSet r[], int x)
{
    if (r[x].ConArg == -1) /* no argument */
        Consequence1 = r[x].Consequence1;
    else if (r[x].ConArg == 0)
        /* one Consequence argument and that is DestIPAddress */
        Consequence1 = r[x].Consequence1 + "DestIPAddress +";
    else if (r[x].ConArg == 1)
        /* two consequence arguments and both DestIPAddress */
        Consequence1 = r[x].Consequence1 + "DestIPAddress +";
        Consequence2 = r[x].Consequence2 + "DestIPAddress +";
    else {
        /* two consequence arguments and one is DestIPAddress and the other is
        SourceIPAddress */
        Consequence1 = r[x].Consequence1 + "DestIPAddress +";
        Consequence2 = r[x].Consequence2 + "SourceIPAddress +";
    }
}

public class Test1 extends MobileTransition implements Serializable
{

    public int seq, caught = 0;
    public Table[] ConseqSet = new Table[1000];
    public Table[] PrereqSet = new Table[1000];
    public SmallTable CorrelatedAlert[] = new SmallTable[1000];
    private hostmachine[] Targets = new hostmachine[10];
    public int ConseqPtr = 0;
    public int PrereqPtr = 0;
    public int CorrelatedPtr = 0;
    public int targetCount = 0;
    public RuleSet[] Rule;

    /* These are the attack events that we know
    about and can detect. */

    public void MyRules()
    {
        Rule[0] = new RuleSet("Sadmind Ping", "OSSolaris", 0, "VulnerableSadmind", 0);
                              "OSSolaris", 1, "GainAccess", 0);
                              "SystemCompromised", 2);
                              2, "ReadyToLaunchDDOSAttack", -1);
        Rule[4] = new RuleSet("Stream_DoS", "ReadyToLaunchDDOSAttack", -1, "DDOSAgainst", 0);
    }

    /* This function fills the Consequence and
    Prerequisite tables with the valid Hyperalerts. */
public void InsertTables(HyperAlert ha)
{
    //insert one item into Consequence table
    ConseqSet[ConseqPtr] = new Table();
    ConseqSet[ConseqPtr].ID = ha.EventID;
    ConseqSet[ConseqPtr].En1 = ha.Consequence1;
    ConseqSet[ConseqPtr].En2 = ha.Consequence2;
    ConseqSet[ConseqPtr].time = ha.EventDate;
    ConseqSet[ConseqPtr].EventName = ha.OrigEventName;
    ConseqPtr++;

    //insert one item into Prereq table
    PreregSet[PreregPtr] = new Table();
    PreregSet[PreregPtr].ID = ha.EventID;
    PreregSet[PreregPtr].En1 = ha.Prerequisite1;
    PreregSet[PreregPtr].En2 = ha.Prerequisite2;
    PreregSet[PreregPtr].time = ha.EventDate;
    PreregSet[PreregPtr].EventName = ha.OrigEventName;
    PreregPtr++;
}

public void InsertCorrTable(int x, int y)
{
    CorrelatedAlert[CorrelatedPtr] = new SmallTable();
    CorrelatedAlert[CorrelatedPtr].id = CorrelatedPtr + 1;
    CorrelatedAlert[CorrelatedPtr].Cause = x;
    CorrelatedAlert[CorrelatedPtr].Result = y;
    CorrelatedAlert[CorrelatedPtr].Events = Integer.toString(x) + " -> " +
        Integer.toString(y);

    //look for matches with events already in the table
    for(int i = 0; i < CorrelatedPtr; i++)
    {
        if( CorrelatedAlert[i].Events.substring(0,4).equals(Integer.toString(y)) )
        {
            //add new event to beginning of i.event
            CorrelatedAlert[i].Events = Integer.toString(x) + " -> " +
                CorrelatedAlert[i].Events;
        }
        else
        {
            if( CorrelatedAlert[i].Events.endsWith(Integer.toString(x)) )
            {
                //add y to end
                CorrelatedAlert[i].Events = CorrelatedAlert[i].Events + " -> " +
                    + Integer.toString(y);
            }
        }
    }
    CorrelatedPtr++;
}

public void PrintTables()
{
    //set up file output stream here.
    String outdataPath = "C://Voyager3-2/MAIDS//correlation//ConseqSet.txt";
    String outdataPath2 = "C://Voyager3-2/MAIDS//correlation//PreregSet.txt";
    DataOutputStream dos = null;

    try {
        File f = new File(outdataPath);
        FileOutputStream fos = new FileOutputStream(f);
        dos = new DataOutputStream(fos);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
BufferedOutputStream bos = new BufferedOutputStream(fos);
dos = new DataOutputStream(bos);
for(int x = 0; x < ConseqPtr; x++)
{
    dos.writeBytes(ConseqSet[x].ID + "," + ConseqSet[x].Enl + "," + ConseqSet[x].En2 + "," + ConseqSet[x].time + "," + ConseqSet[x].EventName + "\n");
}
dos.flush();
dos.close();
}
catch(Exception c) {}
}

try {
    File f = new File(outdataPath2);
    FileOutputStream fos = new FileOutputStream(f);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
dos = new DataOutputStream(bos);
for(int x = 0; x < PreregPtr; x++)
{
    dos.writeBytes(PreregSet[x].ID + "," + PreregSet[x].En1 + "," + PreregSet[x].En2 + "," + PreregSet[x].time + "," + PreregSet[x].EventName + "\n");
}
dos.flush();
dos.close();
}
catch(Exception d) {}

I/PrintTables

/* This function prints the correlated events to file*/
public void PrintCorrTable()
{
    String outdataPath = "C:\Voyager3-2\MAIDS\correlation\RyanResults.txt";
    DataOutputStream dos = null;
    try {
        File f = new File(outdataPath);
        FileOutputStream fos = new FileOutputStream(f);
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        dos = new DataOutputStream(bos);
        for(int i = 0; i < CorrelatedPtr; i++) //look up event id and print the event 'name'
        {
            dos.writeBytes(CorrelatedAlert[i].Events);
            dos.writeBytes("\n");
        }
dos.flush();
dos.close();
}
catch(Exception e) {}
}

/* This function adds the host to a list of hosts known by the agent and
  records how many times the agent has been to that host
*/
public int setTarget()
{
    String tempName = myHost.hostO;
    if(targetCount == 0)
    {
        //add new element to array
        Targets[targetCount] = new hostmachine();
        Targets[targetCount].name = tempName;
        Targets[targetCount].visits = 1;
        targetCount++;
    }
else
{
    for(int i = 0; i < targetCount; i++)
    {
        // look for item; if found, increase visits
        if( Targets[i].name.equals(tempName) )
        {
            Targets[i].visits++;
            targetCount++;
            return (Targets[i].visits);
        }
    }
    // name not found; let's add it in
    Targets[targetCount] = new hostmachine();
    Targets[targetCount].name = tempName;
    Targets[targetCount].visits = 1;
    targetCount++;
}  
return (1);
}

public String agentName()
{
    return "TEST1";
}

public String[] sources()
{
    String[] r = {"Test"};
    return r;
}

public String whichAgent()
{
    String r = "Test1";
    return r;
}

// Main Event Correlation Function
public void check4intrusions()
{
    String input = null;
    String dataPath;
    String tempDate;
    String tempSrcIP;
    String tempDestIP;
    String tempOrigEventName;
    int commal = 0;
    int comma2 = 0;
    int status1, status2 = 0;
    int tempID, tempSrcPort, tempDestPort;
    int beenHere = setTarget();

    if(beenHere < 2) // only read data file once per host for test scenario 1 (off-line).
    {
        MyRules();
        DataInputStream dis = null;
        if(System.getProperty("os.name").startsWith("Windows"))
            dataPath = "C:\Voyager3-2\MAIDS\correlation\abbrevents2 windows.txt";
        else
        {
            if(System.getProperty("os.name").startsWith("Linux"))
                dataPath = "/home/root/abbrevents2.txt";
            else
                dataPath = "C:\Voyager3-2\MAIDS\correlation\abbrevents2.txt";
        }
        try {
            File f = new File(dataPath);
            FileInputStream fis = new FileInputStream(f);
            BufferedInputStream bis = new BufferedInputStream(fis);
}
dis = new DataInputStream(bis);
input = dis.readLine(); //get past header row
while ( (input = dis.readLine()) != null ) {
    //get first field
    comma2 = input.indexOf(' ,');
    tempID = Integer.parseInt( input.substring(0, comma2) );
    //get second field
    comma1 = comma2;
    comma2 = input.indexOf(' ,', comma1 + 1);
    tempDate = input.substring(comma1 + 1, comma2);
    //get third field
    comma1 = comma2;
    comma2 = input.indexOf(' ,', comma1 + 1);
    tempSrcPort = Integer.parseInt( input.substring(comma1 + 1, comma2) );
    //get fourth field
    comma1 = comma2;
    comma2 = input.indexOf(' ,', comma1 + 1);
    tempSrcIP = input.substring(comma1 + 1, comma2);
    //get fifth field
    comma1 = comma2;
    comma2 = input.indexOf(' ,', comma1 + 1);
    tempDestPort = Integer.parseInt( input.substring(comma1 + 1, comma2) );
    //get sixth field
    comma1 = comma2;
    comma2 = input.indexOf(' ,', comma1 + 1);
    tempDestIP = input.substring(comma1 + 1, comma2);
    //get seventh field
    tempOrigEventName = input.substring(comma2 + 1);
    //------------------ compare to rules(5) now ------------------
    try{
        for (int x = 0; x < 5; x++)
        {
            if(Rule[x].IsRule(tempOrigEventName))
            {
                Alert a = new Alert(
                    tempID,
                    tempDate,
                    tempSrcPort,
                    tempSrcIP,
                    tempDestPort,
                    tempDestIP,
                    tempOrigEventName);
                HyperAlert ha = new HyperAlert(a);
                ha.SetConsequence(Rule, x);
                ha.SetPrerequisite(Rule, x);
                //insert ha into table;
                InsertTables(ha);
                break;
            }
        } //end for
    } //end try
    catch(Exception e) {
        System.out.println("Exception thrown in Rules: " + e);
    }
} //end while
}//end try
catch(IOException e) {
    System.out.println("File read error " + e.getMessage());
}
}
finally {
    // close the file
    if (dis != null) {
        try {
            dis.close();
            catch (IOException ioe) {
            }
        }
    }
}
// end if been here

All data now read in from data source tables

Do correlation now on Windows host visit

if (beenHere > 1 & beenHere < 3) &&
(System.getProperty("os.name").startsWith("Windows"))
{
    for(int x = 0; x < ConseqPtr; x++)
    {
        for(int y = 0; y < PreregPtr; y++)
        {
            try{
            }
        }
    }
}
// end correlation

public void setCaught(int number) {
    caught = number;
}
public int getCaught() {
    return caught;
}

public String[] tokenSpec() {
    String[] r = {"FTP_PORT","FTP_PORT_OK"};
    return r;
}
public Token[] unify(Token[] in) {
    System.err.println("T1 starting unify() with:");
    System.err.println("sequence numbers "+in[0].sequence()+" and");
    System.err.println(in[1].sequence()+".");
    if(in[1].sequence() == in[0].sequence()+1)
    {
        System.out.println("Total correlations = " + CorrelatedPtr);
        PrintCorrTable();
        PrintTables();
    }
    catch(Exception e) {
    }
}
// end check4intrusions

}
Token[] out = new Token[1];
out[0] = new Token("col 1", "", "place 1", in[1].sequence(), in);
return out;
}
else
{
    return null;
}
} //Token[] test = new Token[1]; return test;
//---------------------------------------------------------------
} //end class Test1
APPENDIX C: Mobile Transition

/*
MAIDS: Multi-Agent Intrusion Detection System
Iowa State University

MobileTransition:
Abstract superclass of all mobile transition agents.
*/

package MAIDS;
import java.util.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;

abstract public class MobileTransition extends Transition
{
    public abstract String whichAgent();
    public abstract void check4intrusions();
    public abstract int getCaught();
    public abstract void setCaught(int foo);

    /* Inform the agent of a new host it must visit. If list is
    * empty when this call is made, the host will be considered
    * the agent's garage, where no activity takes place.
    */
    public void addHost( VoyagerHost h )
    {
        synchronized( theHosts )
        {
            theHosts.addElement(h.label());
            report();
        }
    }// End Method addHost()

    public void run( VoyagerHost h )
    {
        go( h );
    }// End Method run()

    // Inform the agent of a host it no longer should visit.
    public void delHost( VoyagerHost h )
    {
        synchronized( theHosts )
        {
            theHosts.removeElement(h.label());
            report();
        }
    }// End Method delHost()

    /* Start the transition operating.
    * s: nameserver string, from console
    * src: Vector containing labels for all source places
    */
    public void go( VoyagerHost h )
    {
        nameserver = new VoyagerHost(h);
        running = true;
        mv();
    }// End Method go()

    // The callback function used in for Voyager's moveTo method.
    public void mainloop()
    {
        boolean looping = true;
while( looping )
{
    if( running )
    {
        check4intrusions();
        work();
    }
    else
    {
        // Garbage collect me!
        Agent.of(this).setAutonomous(false);
        return;
    }

    pause(1000+(int)(Math.random()*1000));

    if( mv() )
    {
        looping = false;
    }
}
} // End Method mainloop()

public void parkloop()
{
    boolean looping = true;
    while( looping )
    {
        System.err.println(agentName()+": parked");
        while( theHosts.size() == 0 && running )
        {
            try {Thread.sleep(1000);} catch(Exception e)
            {
            }
        }
        if( running )
        {
            // a host is ready to be visited.
            looping = ! mv();
        }
        else
        {
            // we've been shut down.
            Agent.of(this).setAutonomous(false);
            return;
        }
    }
} // End Method parkloop()

protected void park()
{
    try
    {
        hostIndex = -1;
        Agent.of(this).moveTo(Agent.of(this).getHome(), "parkloop");
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
} // End Method park()

/* Move to the next host.
 * return true iff the move was successful 
 */
protected boolean mv()
{
    int numhosts;
synchronized( theHosts )
{
    numhosts = theHosts.size();
    if( numhosts == 0 )
    {
        if( hostIndex >= 0 )
            park();
        else
            return false; // failure
    }
    else
    {
        hostIndex++;
        if( hostIndex >= numhosts )
            hostIndex = 0;
        myHost = new VoyagerHost((String)theHosts.elementAt(hostIndex));
        System.out.println("moving to host " + theHosts.elementAt(hostIndex) + "\n");
        try
        {
            Agent.of(this).moveTo(myHost.dest(), "mainloop");
        }
        catch( Exception e )
        {
            System.err.println("Mobility exception: "+e);
            return false;
        }

        return true;
    } // end of if there are hosts
} // end of sync block (theHosts)
return true; // actually, can't get here
} //---------------------------End Method mv()---------------------------
protected void report()
{
    System.err.println("hosts for this agent:");
    for( int i =0; i < theHosts.size(); i++ )
        System.err.println(theHosts.elementAt(i));
} //----------------------------End Method report()------------------------

private Vector theHosts = new Vector();
private int hostIndex = 0; // forces agent to be explicitly parked
private boolean ready = false;
} //------------------------------End Class MobileTransition.java------------------------