SPARTA

A Mobile Agent based Intrusion Detection System

Christopher Krügel*
Distributed Systems Group
Technical University Vienna
chris@infosys.tuwien.ac.at

Thomas Toth
Distributed Systems Group
Technical University Vienna
ttoth@infosys.tuwien.ac.at

Abstract The large number of machines with different operating systems and applications in an enterprise network makes it very difficult for the system administrator to close all security holes and install the latest OS and software patches. When the network is connected to the Internet and services are remotely available they become a potential target for hackers. As the number of security related incidents is constantly increasing at an alarming rate the need for automated tools to detect intrusions becomes evident. Such tools are called intrusion detection systems (IDS).

We present Sparta, a system that allows to detect security policy violations and network intrusions in a heterogeneous, networked environment. We have designed a pattern language in order to express intrusions (i.e. offending event patterns) in a declarative manner. This allows to specify what to detect instead of how to detect. A fully distributed approach to find the given patterns is presented as well. We use mobile agents to correlate event data instead of moving the whole information to a central location. This increases the fault tolerance and scalability of our system.

Keywords: Intrusion Detection, Mobile Agents, Pattern Specification Language, Event Correlation, Network Security

*Contact Author

Introduction

Virtually every organization depends on sensitive data which has to be protected against unauthorized access. Such data is often stored on machines which are remotely available over a network. The growth of the Internet has caused an increase of the size of individual networks as well as an increase of transported traffic. This makes it extremely difficult to manually manage and protect valuable assets. Combined with an alarming rise of attacks and hacking attempts, organizations need tools like intrusion detection systems (IDS) to enforce security and detect hacking attempts.

Sparta (an acronym for Security Policy Adaptation Reinforced Through Agents) is the name of a system architecture which is capable of monitoring a network to detect network intrusions and security policy violations. The system monitors local events at hosts which are connected by a network, relates them and provides an interface where the user can query the gathered information. This makes it possible to apply our design to a broad range of applications and use it for a number of network related tasks, ranging from network management to intrusion detection.

The contribution of this paper is the description of an architecture to collect and relate distributed data in an efficient way by using mobile agents and its application to network intrusion detection. In contrast to traditional designs where data is gathered and analyzed at a central location, the application of mobile agents allows distributed analysis. This approach improves the scalability and increases the fault tolerance in our opinion.

The paper is structured as follows. First, we present an overview of the system's functionality and its global architecture. Then we describe our way of specifying distributed patterns and present an efficient implementation to detect them with mobile agents. Finally, we relate our work to other approaches and briefly conclude.

1. Functional Description

Sparta is an architectural framework which allows to identify and relate interesting events that may occur at different hosts on a network. A single event is described by specifying appropriate values for its attributes. A number of events can be connected by defining temporal or spatial relationships between them or imposing certain constraints on their attributes thereby creating a pattern. In order to deal with complex patterns and systems, it is not sufficient to select events based on content alone. It is necessary to consider multiple events at the same time and deduce knowledge that is beyond the scope of an individual event. The process of detecting a set of events with given properties is called *correlation*.

This general correlation capability allows the Sparta architecture to be used for different distributed applications, ranging from network security to network management implementations. We currently build a security policy and ID application based on our design.

The basic functionality can be described as follows. Interesting events are locally collected and stored. The collection of all local information can be considered as a distributed database with horizontal fragmentation. For each relation (i.e. event type), the tuples (i.e. actual events) are stored at different locations. A user may issue queries in our Event Query Language (EQL) to search for a set of events that fulfill his desired constraints. In addition to this the system can also be used to gather statistical information. It is possible to find the number of pattern instances at each host and to calculate the maximum or minimum for event attribute values as well as their sums over a set of hosts. The query is carried out by mobile agents which return their results to the user.

For our intrusion detection system, a failed authentication attempt or the start of a root shell might be examples of interesting events. Sparta allows to count the number of failed telnet logins for a certain user throughout the network (to detect distributed door knob rattling attempts) or to find tree-like connection patterns between hosts (to identify a spreading worm). It is important to notice that event correlation might yield information that is impossible to gain by just looking at a single node. Consider an intruder who tries to cover his tracks by performing several consecutive telnet logins (i.e. producing a **telnet chain**). This is an often observable behavior that exploits the fact that different machines are administered by different people and don't have synchronized local clocks. Tracing an attacker by having to look at all these logfiles is rather difficult. On each local machine only a simple ingoing and outgoing connection is noticed but when looking at the entire network the offending pattern becomes evident. GrIDS (Staniford-Chen et al., 1996) is a well known ID system which bases its detection solely on looking for connection patterns but uses a different mechanism to collect and relate data.

2. System Architecture

The system consists of a set of hosts connected by a network where each node has the following components installed (see Figure 1).

- Local event generator (sensor)
- Event storage component
- Mobile agent platform
- Agent launch and query unit (optional)

To ensure a high degree of portability, most of the components are written in Java. The agent platform itself bases on Gypsy (Lugmayer, 2000), a Java agent system which has been developed at our department. Parts of the sensor and security related code, which needs direct access to system resources not provided by Java, are written in C/C++ and connected via the Java Native Interface (JNI).

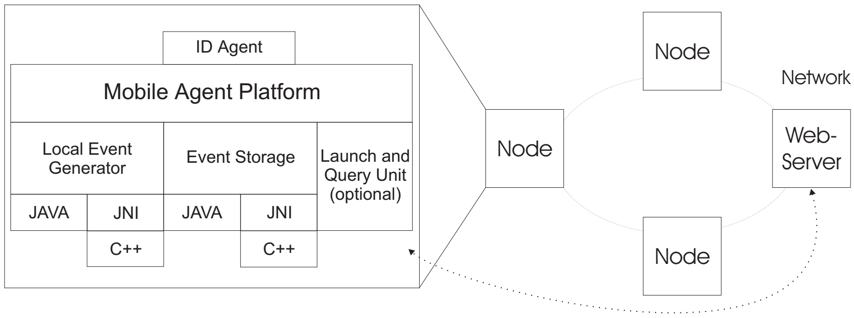

Figure 1. Sparta Architecture

The local event generation is done by sensors which monitor interesting occurrences on the network (network based) or at the host itself (host based detection). The exact types of events and their attributes as well as the implementation of the sensors are mainly determined by the application's needs. The type of an event is represented by the type of the class in the implementation (i.e. Java class), with the event's attributes being stored by the members of the corresponding class. It is possible to extend an event by subclassing from an existing one and add the desired additional information. This allows to write patterns which relate high level events and have the system automatically consider all actual instances (i.e. subclasses) of such generic events.

Sensors store their generated data in a local data storage component, preferably a database. The data storage component must be able to support the inheritance relationship of events. When queries specify parent class events, derived events have to be returned as well.

The mobile agent subsystem is responsible for providing a communication system to move the state and the code of agents between different hosts and for providing an execution environment for them. Additionally, the system has to provide protection against security risks involved when utilizing mobile code (see Section 5 for more details). An important task of the agent subsystem is the provision of a directory service. When agents have to look for event patterns, they need to access a list of all hosts with an installed agent platform. The agent platform also provides clock synchronization with a maximum guaranteed deviation. This is needed to be able to temporally relate events at different nodes.

The user interface allows users to specify queries and claim the results. The agent launch and query unit initiates the launch of appropriate agents and provides a way for them to communicate back their results. Queries are written in a language called Event Query Language (EQL), which we have developed to conveniently specify patterns that reflect a security violation. This is described in more detail below in Section 3.1. The user interface itself is realized as a web interface using HTML and JavaScript on the client side and Servlets on the server side. The communication between the client and the server is secured by using SSL connections. This setup allows a user to access the system via a standard browser from any computer that needs no Sparta components installed.

3. Pattern Specification

The design of our pattern specification language is guided by two conflicting goals. The first goal states that the language should be as expressive as possible. It would be desirable to allow the description of complex relationships between events on different hosts using regular or tree grammars. Unfortunately, the evaluation of complex patterns makes it necessary for each local host to send a huge amount of data to a central site. This conflicts with the second goal, which demands that the amount of data that has to be transferred between hosts should be as small as possible. When a system uses mobile code (i.e. mobile agents), it should aim at performing flexible computation remotely at the location where the interesting data is stored instead of abusing agents as simple data containers.

When the interesting patterns do not change frequently, it would be desirable to wire them directly into local components at each host. For our application, users intend to specify many different patterns and perform a lot of ad-hoc queries. Therefore, the application of mobile code is reasonable.

The basic building block of a pattern is a set of local events. One can specify a list of events on a local host by enumerating them and imposing certain constraints on their attributes. A constraint can have two different formats. One format allows to relate an event attribute with a constant value, using one of the standard logical operators or one of our self-defined ones (`in`, `range`). The other format allows to relate an attribute of one event with another attribute of the same or a different event, again using the full range of operators. This allows to select a number of events with a common context. A connection between events on different hosts is established by connection events.

Definition:

A pattern P*, relating events that occur at n distinct hosts, consists of n sets of events, one for each node.*

Definition:

A set of events S_A *at host* A *is linked to a set of events* S_B *at host* B*, iff* S_A *contains a send event and* S_B *contains the corresponding receive event. When node* A *opens a channel to* B *for data transmission (e.g. open a TCP connection, send a UDP packet, send an Ethernet frame), a pair of corresponding events (send at* A*, receive at* B*) is created.*

Definition:

Pattern P *is valid, iff the following properties hold.*

1. *Each set of events is at least linked to one other set.*
2. *Every set except one (called the root set) contains exactly one send event. The root set contains no send event.*
3. *The connection graph contains no cycles. The connection graph is built by considering each event set as a vertex and each link between two sets as an edge between the corresponding vertices.*

These definitions actually only allow tree-like pattern structures (i.e. the connection graph is a tree), where the node with the root set is the root of the tree. Although this restriction seems limiting at a first glance, most desirable situations can still be described. Usually, activity at a target host only depends on events that have occurred earlier at several other hosts. This situation can easily be described by our tree patterns where connection links from those several hosts end at the root node. The opposite case, where events on two different nodes both depend on the occurrence of a single event at a third node is more difficult. In this case, the connection links do not end at the root node, but have their origin there. Such a situation cannot be directly expressed in our pattern language (as the root node set would contain two send events). Nevertheless, an application might split the original, illegal pattern into subpatterns (each representing a legal tree like structure) and relate the results itself. This allows to define arbitrary complex patterns at the expense of performance and network traffic. The major advantage of the proposed limitation is the possible implementation of an efficient search algorithm (for details, see Section 4) which transfers as little data as possible over the network.

Additionally, our query language allows to combine pattern specifications with the possibility to extract statistical data. A pattern instance is a set of events that satisfy the constraints of a valid pattern. Obviously, it might be possible that a single pattern is satisfied by more than one event set. Two event sets are said to be distinct, if they contain at least one distinct event element. An event element can be uniquely identified by its timestamp and the host, where it occurred. Statistical data can be computed for the set of all distinct instances of a given pattern. One can obtain the number of elements in that set (i.e. the number of valid instances) or the maximum or minimum values for the number of instances at each host. Additionally, one can query attribute values of a certain single event of the pattern. The sum, maximum or minimum values over an attribute may be calculated.

3.1. Event Query Language

This section describes the syntax and semantics of our Event Query Language (EQL) in more detail. We omit the complete language grammar, instead we gradually introduce the language by giving explanations on several examples.

A query in our language looks similar to an SQL select statement and is written as follows.

```
SELECT results FROM nodes WHERE conditions
```

The *results* section is used to define the type of information the user is interested in. The operator `COUNT` can be used for patterns and returns a list of all nodes with the number (i.e. count) of found pattern instances at each one. In addition to that, the operators `SUM`, `MAX` and `MIN` may be used for patterns as a whole or for an attribute of a single event (marked by an attribute identifier). When used for patterns, these operators return the sum, the maximum or the minimum number of detected pattern instances per node, respectively. When used for an event attribute, the sum or the extreme value (maximum/minimum) for a certain attribute value over all instances is returned.

The *nodes* section is used to assign an identifier to each node that is later used in the pattern definition. Additionally, one can impose restrictions on each node to have the agents only consider a limited set of actual hosts.

```
SELECT COUNT FROM host_1 range (10.1.17.0, 10.1.17.255)
# return the number of pattern instances for each host which is on the
# 10.1.17.* subnet (i.e. has an IP address between 10.1.17.0 and
# 10.1.17.255 (inclusive)
```

The *conditions* section specifies the pattern. It consists of a list of event sets, one for each node that appears in the node section. The event set is a list of identifiers, each describing an event. In order to be able to specify statistics operations on event attributes, one can assign identifiers (i.e. a label) to each of them. Two predefined labels called `send` and `receive` are used to identify the send and receive events, respectively, for linking event sets (see Section 3).

Each event can optionally be defined more precisely by constraints on the event's attribute values. These attribute values can be related to constant values or to variables by standard operators (`=, !=, <, >, >=` and `<=` with their usual semantical meaning) or by a `range` or an `in` operator as defined below.

x range $(x_0, x_1) \leftrightarrow x_0 \leq x \leq x_1$
x in $(x_0, x_1, ..., x_n) \leftrightarrow \exists$i (0 $\leq$ i $\leq$ n) and x = x_i

A variable is defined the first time it is used. One must assign a value (bind an attribute value) to each defined variable exactly once. Each variable may be used arbitrarily often as a right value in constraint definitions. The scoping of variables is global and it inherits its type from the attribute which is used to define it.

With these explanations, we may introduce the syntax (in BNF) of the conditions section (all identifiers represent strings).

```
conditions  : {event set}+
event set   : node-identifier '{' {event}+ '}'
event       : [connection] event-identifier '[' {constraint ';' }* ']'
constraint  : assignment | [label] relation
assignment  : '$'variable-identifier '=' ( attribute-id | constant )
relation    : attribute-id operator ['('] {value ',' }* value [')']
value       : constant | '$'variable-identifier
operator    : '<' | '>' | '<=' | '>=' | '=' | '!=' | in | range
connection  : 'send('target-id'):' | 'receive('source-id'):'
label       : label-identifier':'
constant    : numerical constant | string constant
```

The following example shows a classical telnet chain pattern that describes a connection from Node1 to port 23 at Node2 and from there to port 23 at Node3. Node3 describes the root node set (i.e. has no outgoing send event).

```
Telnet Chain:

Node1 {
  send(node2): tcp_connect []
}
Node2 {
  receive(node1): tcp_accept [ port = 23; ]
  send(node3): tcp_connect []
}
Node3 {
  receive(node2): tcp_accept [ port = 23; ]
}
```

4. Pattern Detection

Usually, patterns are discovered by gathering event data from distributed nodes at a central host where pattern matching algorithms distill the desired information. Our approach differs from the usual setup by detecting patterns in a fully distributed way using mobile agents. Mobile agents roam the network to search for suspicious events to start a more detailed investigation. When an agent spots the mark of a possible intrusion, it decides which data to carry with it on its next hop and which place to visit next.

The advantages of such a pattern detection scheme base on the fact, that no central entity is needed for data correlation. This increases the fault tolerance and robustness of the system, which is especially important for security relevant systems. When the host where a centralized IDS performs its correlation is taken out of action (e.g. by a DoS attack) the detection mechanism is actually blinded. When an attack renders some hosts in the network unavailable, agents can still search the remaining ones for signs of intrusions. Even when an attacker takes over a few hosts and manages to modify the agent platform in a way that it delivers wrong data (simply bringing it down is suspicious by itself), only intrusions where parts of the pattern occur at the compromised hosts are not detectable any more. The remaining system is still working and able to detect security violations.

Our approach also improves the scalability of the system because new hosts on the network won't automatically cause additional traffic to a single existing server machine. While traditional approaches like hierarchical installations and redundant servers allow to process more traffic than a single machine, a distributed approach is still desirable. We think that we can exploit the locality of network accesses. Most connections in large companies are between machines of the same department (like references to internal web servers or file shares) while connections between departments are rare. This allows agents to look for patterns in small areas and then move on. In a system with a central root node, all traffic would need to be forwarded (even with prefiltering and reduction over several hierarchies) to it.

The detection is done in the following way. An agent is started by the user interface with a given pattern (representing a security violation) that it has to look for. It starts its task by contacting the directory service to obtain a list of all hosts with an installed agent platform that match the constraints given in the pattern's `FROM` clause. These nodes are then visited in arbitrary order.

When an agent arrives at a host, it looks for events that fulfill the constraints given for the root node of the pattern it is currently investigating. In the case of the telnet chain introduced above, the agent would have to look for accepted TCP connections at port 23 (see Step 1 of Figure 2). The result of this process are a number of events (representing different instances of the pattern) which satisfy the root node constraint.

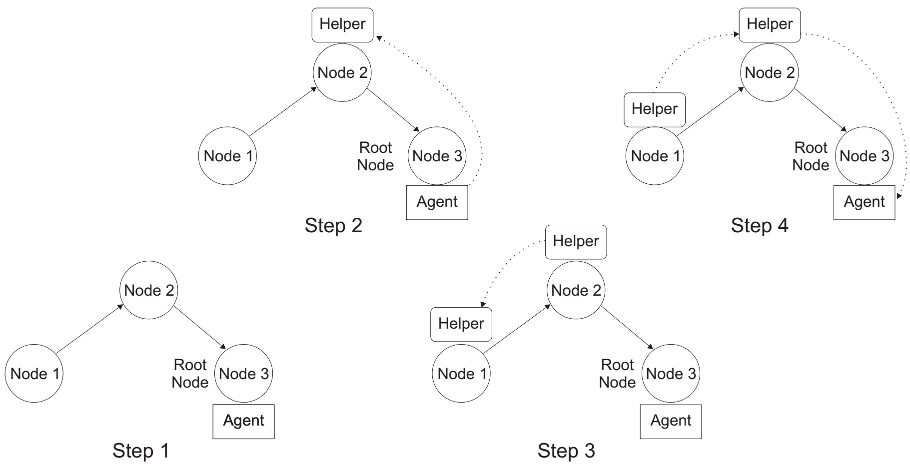

Figure 2. Pattern Detection

When no events are found, the agent immediately continues its journey. Otherwise, all receive events are identified. In the telnet chain case only receive events exist but more complex patterns are possible. For the distributed detection algorithm, a receive event is important because it establishes a relationship between nodes which is used to select a promising next place where an agent should look further. For each receive event, a helper agent is spawned which follows the link to the host with the corresponding send event (see Step 2 of Figure 2). By using the send event (determined by the receive event of the node the agent was coming from) it looks for events which fulfill the current part of the pattern. In the case of our telnet chain example, the agent already knows that the send event to Node 3 exists (as it is coming from there) and now searches for another accepted TCP connection at port 23 (from Node 1).

When the local event set contains receive events itself, the process recursively repeats by having the agent spawning helper agents and waiting for their return (see Step 3 of Figure 2). When the helper agents return, they report their findings (i.e. pattern instances) back to the agent waiting at the originating node. Helper agents only move over a single hop to a remote node and eventually return back. When pattern instances are returned, the waiting agent processes them (e.g. match variables or perform statistical evaluation) and eventually continues. When all helper agents have returned a pattern might be detected by the agent waiting at the root node as all information is available (see Step 4 of Figure 2).

Usually, only a small amount of data has to be transferred as it is not necessary to transport all pattern instances themselves but merely time stamps or single attribute values which have been assigned to variables.

Variables are treated in the following way. When a variable has already been bound to a value, it is straightforward to use this value directly for the attribute's constraint. This is the case when a value is assigned to a variable at a node which is closer to the root of the connection graph than the node, where it is used. On the other hand, when a constraint depends on a variable which has not been resolved yet, one has to temporary ignore that constraint. First, all actual bindings of the variable have to be determined and are then matched against the instances where the variable has not been assigned yet. This allows to filter out pattern instances which do not satisfy the previously ignored constraint. Notice that this may cause agents to transfer unnecessary pattern instances but the runtime complexity is still linear with regard to the number of send/receive events.

Instead of having a single agent visit all nodes sequentially, the task could be parallelized easily by partitioning the set of interesting hosts. Each partition is visited by a dedicated agent which all have to agree on a destination node, where they meet and merge their results.

We have installed a first prototype version at our department's network as universities are traditionally favourite targets of hackers. We are currently able to detect about a hundred local events (by looking for well known attack signatures and network connections) and a dozen distributed patterns. The results are promising as a couple of incidents have already been detected. The network overhead of the travelling agents is negligible and the processing overhead at each node is reasonably low.

5. Security

Mobile code introduces a number of security issues that our design has to deal with. Especially when building systems for security sensitive applications (like our intrusion detection system), it is important not to introduce new vulnerabilities by the security monitoring tool itself. The security threats to mobile agents are classified by four categories, namely agent-to-agent, agent-to-platform, platform-to-agent and other-to-agent.

- *Agent-to-agent threats* describe the set of attacks, where one agent exploits the vulnerabilities of another agent. In Sparta, agents only communicate with helper agents which they have previously spawned. This is done locally via message passing. As arbitrary interagent communication is prohibited, possible vulnerabilities cannot be exploited and agent-to-agent attacks can be prevented.
- *Agent-to-platform threats* describe attacks, where an agent performs malicious activities against its environment (i.e. platform). To prevent these kind of attacks, the runtime permissions of agents are rigorously restricted. They are not allowed to access resources directly. Instead, agents gain information by querying the data storage component. A configurable security manager and permission tables make it possible to restrict the available information from the storage component individually for each agent.
- *Platform-to-agent threats* describe situations, where the platform compromises the agent's security. This sort of threats is extremely difficult to defend against when agents need unrestricted movement around the network (Jansen and Karygiannis, 1999) and Sparta has no special mechanisms to defend against such attacks.

  In contrast to a central server system, we still have the advantage that even if a single node is compromised, all patterns which do not touch this host are still detectable. When the host with the ID system is compromised by an attacker, usually the local detection capability is rendered useless. This is not a problem specific to a mobile agent based solution but affects all intrusion detection systems.
- *Other-to-agent attacks* involve threats against agents performed by external entities while they are in transit over the network (e.g. eavesdropping

or tampering). Sparta uses an asymmetric (public/private key pair) cryptosystem to secure agents when they are transferred over the network. The agent code is signed and can be authenticated before it is executed. In order to manage the asymmetric cryptosystem, a Public Key Infrastructure (PKI) is provided.

6. Related Work

The idea of correlating events which occur at different places in a network and to formalize patterns to describe such correlations is not new. The Complex Event Processor which is developed at Stanford University is capable of correlating causally and temporally related events. It bases on the theory of partial ordered multisets (Pratt, 1986) and is used for intrusion detection (Perrochon et al., 2000) and network management (Perrochon et al., 1999). Patterns are described using the Rapide Pattern Language (RAPIDE, 1997). The difference between our approach and their work is the fact, that we use mobile agents to perform the pattern detection in a distributed fashion without any central server. In contrast to that, they collect data from different client sites and process it at a server. Rapide has clearly influenced our work as their system as well as ours try to correlate generic events and target a broad spectrum of applications.

State-of-the-art ID systems like EMERALD (Porras and Neumann, 1997), NStat (Kemmerer, 1997) or AAFID (Balasubramaniyan et al., 1998) can gather and relate data from different sources. In contrast to our distributed design, they have a hierarchical architecture where sensors located at different hosts collect data and send it to a central entity where events are related. Unlike Rapide and our design, they completely focus on intrusion detection events and are not applicable to different domains. The same is true for network management software (e.g. HP OpenView (Sheers, 1996)), which usually offers simple filtering and aggregation mechanisms.

Commercial intrusion detection systems like Network Flight Recorder (NFR, 2001), Real Secure (RealSecure, 2001) or Net Ranger (NetRanger, 2001) perform their analysis on packet level by monitoring network traffic. This allows only simple correlation, but their output can be used as basic events for our system.

Only a few intrusion detection systems have tried to use mobile agents so far. While IDA (Asaka et al., 1999) mainly uses them to trace a possible attacker back to its origin, Micael (de Queiroz et al., 1999) pursues a more ambitious aim where each system component is realized as a mobile agent. Unfortunately, only a high-level system design has been presented and no details have followed so far. The possible advantages of mobile agents in intrusion detection systems are summarized in (Jansen et al., 1999) and (Krügel and Toth, 2001).

7. Conclusion

Relating distributed events and deducing knowledge from different hosts is especially important in the field of network management and intrusion detection.

We present a solution, where mobile agents perform the task of correlating data in a fully decentralized manner. Designing a distributed system always includes the goal of creating a scalable solution. In order to prevent a tremendous increase in network traffic, the expressiveness of our pattern description language had to be slightly restricted. This allows an efficient detection algorithm and a fault tolerant and scalable system design.

References

Asaka, M., Taguchi, A., and Goto, S. (1999). The implementation of ida: An intrusion detection agent system. In *Proceedings of the 11th FIRST Conference*.

Balasubramaniyan, J. S., Garcia-Fernandez, J. O., Isacoff, D., Spafford, E., and Zamboni, D. (1998). An architecture for intrusion detection using autonomous agents. In *14th IEEE Computer Security Applications Conference*.

de Queiroz, J. D., da Costa Carmo, L. F. R., and Pirmez, L. (1999). Micael: An autonomous mobile agent system to protect new generation networked applications. In *2nd Annual Workshop on Recent Advances in Intrusion Detection*.

Jansen, W. and Karygiannis, T. (1999). Mobile agents and security. Special Pub. 800-19, NIST.

Jansen, W., Mell, P., Karygiannis, and Marks, D. (1999). Applying mobile agents to intrusion detection and response. Interim Report (IR) 6416, NIST.

Kemmerer, R. A. (1997). A model-based real-time network intrusion detection system. Technical report, Computer Science Dep., University of California Santa Barbara, November.

Krügel, C. and Toth, T. (2001). Applying mobile agent technology to intrusion detection. In *ICSE Workshop on Software Engineering and Mobility*.

Lugmayer, W. (2000). Gypsy: A component-based mobile agent system. In *8th Euromicro Workshop on Parallel and Distributed Processing (PDP 2000)*, Rhodos, Greece.

NetRanger (2001). NetRanger. http://www.cisco.com/univercd/cc/td/doc/product/iaabu/netrangr/.

NFR (2001). Network Flight Recorder. http://www.nfr.net/.

Perrochon, L., Jang, E., and Luckham, D. C. (2000). Enlisting event patterns for cyber battlefield awareness. In *ARPA Information Survivability Conference and Exposition (DISCEX'00)*.

Perrochon, L., Kasriel, S., and Luckham, D. C. (1999). Managing event processing networks. Technical Report CSL-TR-99-877, Stanford Computer Systems Laboratory.

Porras, P. A. and Neumann, P. G. (1997). Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th NIS Security Conference*.

Pratt, V. (1986). Modelling concurrency with partial orders. *Int. Journal of Parallel Programming*, 15(1):33–71.

RAPIDE (1997). *Rapide 1.0 Pattern Language Reference*. Stanford University.

RealSecure (2001). RealSecure. http://www.iss.net/customer_care/resource_center/product_lit/.

Sheers, K. R. (1996). HP OpenView Event correlation. *Hewlett-Packard Journal*.

Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., and Zerkle, D. (1996). Grids - a graph based intrusion detection system for large networks. In *Proceedings of the 20th National Information Systems Security Conference*, volume 1, pages 361–370.