Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 79 (2006) 1–14
The Journal of Systems and Software
www.elsevier.com/locate/jss

Towards the automatic generation of mobile agents for distributed intrusion detection system

Yanxin Wang *, Smruti Ranjan Behera, Johnny Wong, Guy Helmer, Vasant Honavar, Les Miller, Robyn Lutz [1], Mark Slagell

Department of Computer Science, Iowa State University, Ames, IA 50011, United States

Received 28 October 2003; received in revised form 3 August 2004; accepted 3 August 2004

Abstract

The Mobile Agent Intrusion Detection System (MAIDS) is an agent based distributed Intrusion Detection System (IDS). A disciplined requirement engineering process is developed to build MAIDS. The starting point is a high level description of intrusions expressed as Software Fault Trees (SFTs). Then the SFTs are translated to Colored Petri Nets (CPNs) that specify the IDS design. Subsequently, the CPNs are implemented as software intrusion detection agents in the MAIDS agent system. By using SFT and CPN as the theoretical underpinnings, the design and implementation of MAIDS can be verified and the design and implementation errors can be substantially reduced.

This paper presents a tool that automatically translates CPNs that specify IDS design into software intrusion detection agents in MAIDS. Together with the translator we have developed to convert SFTs that model intrusions into the CPN for IDS design, this tool can automatically generate intrusion detection software agents from a high level description of intrusions.

Keywords: Intrusion detection; Software agents; Compiler; Software Fault Tree; Colored Petri Net

1. Introduction

The widespread use of the Internet and the easy availability of high-speed access have greatly increased the frequency and the range of attacks that are being launched against computer systems across the world. Tools for launching attacks are easily available on the web, thus reducing the level of computer expertise required to launch attacks. Moreover, with the Internet now being used for a wide range of commercial and military purposes, the damage caused by such attacks has also increased dramatically. An Intrusion Detection System (IDS) not only helps the administrators to detect intrusions and limit damages, but also helps to identify the source of attacks, which sometimes acts as a deterrent especially in case of insider attacks.

Currently, IDSs are often designed using ad hoc methods, and they are prone to design and implementation errors. This is a critical problem for IDSs where a faulty design and implementation might give the end user a false sense of security. We propose to use formal methods to reduce design and implementation errors of IDSs. The Mobile Agent Intrusion Detection System (MAIDS) is a mobile agent based distributed IDS developed at the Information Security Lab in Iowa State University (Information Security Lab, 2003). The biggest advantage of MAIDS, which distinguishes it from other

* Corresponding author. Tel.: +1 515 2944377; fax: +1 515 2940258.

E-mail addresses: wangyx@cs.iastate.edu (Y. Wang), bsmruti@cs.iastate.edu (S.R. Behera), wong@cs.iastate.edu (J. Wong), ghelmer@cs.iastate.edu (G. Helmer), honavar@cs.iastate.edu (V. Honavar), lmiller@cs.iastate.edu (L. Miller), lmiller@cs.iastate.edu (R. Miller), rlutz@cs.iastate.edu (R. Lutz).

[1] Also affiliated with Jet Propulsion Lab/Caltech. The author's research is supported in part by NSF grants 0204139 and 0205588.

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.08.017

IDSs, is that it is designed, implemented and verified using formal method, with mobile agent code automatically generated from formal specification of intrusions. The MAIDS project uses Colored Petri Net (CPN) and Software Fault Tree (SFT) as the formal models in developing IDS.

CPN has been extensively used to model complex and distributed systems (Jensen, 1992). We propose to use CPN to model IDS design since CPN can model the gathering, classification and correlation of activities of IDS very well. There also exist CPN tools (such as, Design/CPN, Denmark CPN Group at University of Aarhus, 2003) which allow us to simulate attacks in order to validate the correctness of an CPN design of IDS.

However, a way is needed to ensure that a correct design is translated into a correct implementation. If the CPN to intrusion detection software agents translation is done manually, it is difficult to make such an assertion. If we use an automated tool, which makes the translation based on some sound theoretical reasoning, we can assure the implementation does reflect the correct design. In this paper, we explain an approach to translate CPN for IDS design to intrusion detection agents, and present a compiler to make the translation.

The rest of the paper is organized as follow. Section 2 contains a brief outline of several distributed IDSs that have been proposed. We will also introduce our previous work using a formal method for describing attacks. Section 3 gives an overview of the MAIDS agent system design. Section 4 describes the design of the compiler. Section 5 contains the implementation of the compiler. Section 6 presents the experiment for the compiler and the compiler-generated code. Section 7 concludes our paper and gives future directions.

2. Background and our previous work

In this section, we will review the distributed IDSs. We will also briefly explain our previous work using SFT to specify intrusions and using CPN as intrusion detection design template.

2.1. Distributed intrusion detection systems

IDS should be able to correlate data from different machines to detect distributed attacks. Some attackers use several machines to coordinate an attack. For example, in FTP Bounce Attack, three machines are used (CERT Coordination Center, 2003a). Recently some large scale coordinated attacks, such as Distributed Denial of Service (DDoS) attacks, increase dramatically (Dittrich, 2003). In order to detect these distributed attacks, distributed IDS is needed.

Many distributed IDSs have been proposed. Of these systems, the mobile agent based distributed IDSs are very promising for the following reasons (Slagell, 2001):

Reduction of data movement: this ensures that processing of the data can be done at the place where it is gathered. This increases the efficiency of the IDS and reduces the time lag.
Load-balance: mobile agents can spread the workload of the IDS over a number of machines.
Flexibility: agents can operate independently of each other. So individual pieces can be removed, modified and improved while the system continues to function.
Fault-tolerance: the fact that agents can operate independently also implies that the system can continue to work even when one agent is destroyed in an attack. This makes the IDS fault tolerant.
Detection of distributed attacks: the use of mobile agents makes it easier to correlate and detect distributed attacks.

The Java Agents for Meta-Learning (JAM) project at Columbia University uses a secured agent infrastructure for continuous learning of fraud and intrusion patterns (Lee et al., 1999). This system uses two kinds of agents: local fraud detection agents learn how to detect fraud and provide intrusion detection services with a single corporate information system, and the other meta-learning agents combine the collective knowledge acquired by individual local agents. The JAM project focuses on using intelligent learning algorithm to generate intrusion detection rules, whereas our project focuses on using formal method to model intrusion and intrusion detection and automatically generate intrusion detection agents.

The Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) project was developed by SRI International (Porras and Neumann, 1997). EMERALD uses a hierarchical approach that provides three levels of analysis performed by a three-tiered system of monitors: service monitors, domain monitors, and enterprise monitors. These monitors have the same basic architecture: a set of profiler engines (for anomaly detection), signature engines (for signature analysis), and a resolver component that integrates the results generated from the engines. EMERALD is a surveillance and response architecture oriented toward the monitoring of distributed network elements. Agents in our project also have a hierarchical architecture: low level agents are for data collection, middle level agents are for data correlation, and high level agents are for intrusion alarm generation. The main difference is we use formal methods to model intrusion and intrusion detection in the development of our IDS.

The Autonomous Agents For Intrusion Detection (AAFID) project uses a flexible and distributed IDS infrastructure (Spafford and Zamboni, 2000). There

are agents, transceivers, monitors and user interfaces. AAFID uses agents as their lowest-level element for data collection and analysis and employs a hierarchical structure to allow for scalability. All the agents in a host report their findings to a single transceiver. Transceivers control these agents and report their results to monitors. Monitors can be organized in a hierarchical structure that a monitor may in turn report to a high-level monitor. In AAFID, though the agents can communicate with each other, they are architecturally and conceptually independent. It is the responsibility of the monitors to make sense of the information they get from the various agents. Our agents also have a hierarchical architecture, but our agent code is automatically generated from SFT description of intrusions.

Distributed Intrusion Detection System (DIDS) was developed by University of California, Davis (Snapp et al., 1991). This system focuses on extending the intrusion detection from single segment of network to arbitrary large networks. The architecture for the system includes a host manager for monitoring each host, a LAN manager for monitoring each LAN in the system, and a central manager which is in a central secure host and receives event reports from hosts and LAN managers, processes these reports, correlates events and generates intrusion alarms. In DIDS, the monitoring and analysis tasks are distributed among the hosts and LAN managers. The central manager gets distributed audit data and correlates events, so it views the entire system and detects intrusions involving multiple hosts and networks. DIDS is similar to our system in that we all use distributed sensors to monitor the hosts and the networks, and a central manager to generate alarms. The difference is DIDS uses expert system and rules to match intrusion patterns and our system uses SFT to model intrusions and CPN to model intrusion detection.

Common Intrusion Detection Framework (CIDF) aims at enabling different intrusion detection and response components to interoperate and share information (Intrusion Detection Hotlist, 2003). The CIDF is a standard proposed by the Information Technology Office of the Defense Advanced Research Projects Agency, University of California—Davis, Information Sciences Institute, Odyssey Research, and others. CIDF views IDSs as consisting of discrete components that communicate via message passing. CIDF consists of four kinds of IDS components: Event Generators, Event Analyzers, Event Databases and Response Units. CIDF is a big step towards getting different IDS to interoperate with each other. Since intrusions are taking on a grander scale, many attacks can be orchestrated over a wide area network, and over a long period of time. As IDSs are developed by various vendors and deployed to various locations, it is very important for the IDSs to be able to share information, infer possible distributed and coordinated intrusion, and warn others about impending attacks. Our system uses an architecture that resembles what describes in CIDF in that we use low level agents to collect distributed data, which correspond to Event Generators in CIDF; We use middle level agents to correlate events, which correspond to Event Analyzers in CIDF; we use high level agents to generate alarms, which correspond to Decision and Response Unit in CIDF. And we also use database backend to store events, which is like Event Database in CIDF. CIDF only gives a framework for IDS, and no special method is given to detect intrusions. Our MAIDS use SFT to describe intrusion, use CPN to specify intrusion detection template and generate intrusion detection agents automatically using the compiler described in this paper.

2.2. Use of SFT for specifying intrusion

The SFT used to model the intrusions is a backward search (Helmer et al., 2001). It begins with an intrusion as the root node and traces back through the possible parallel and serial combinations of events that caused such an intrusion. Basic SFT is static, and it cannot represent time orders of events. However, the time relationships among events are critical in intrusion detection domain. To precisely model intrusions, we extend basic SFT with time constraints (Wang et al., submitted for publication).

SFT analysis of intrusions results in a number of benefits in IDS design (Helmer et al., 2001): SFT enables structured analysis of intrusions, including severity and probability analysis; SFT analysis assists the IDS development process by modeling intrusions, helps to identify priorities for development, and specifies requirements for an IDS; and SFT models of intrusions may help to identify appropriate countermeasures.

2.3. Use of CPN for specifying design of the IDS

CPN is a well-documented and frequently used abstraction for modeling complex and distributed systems. It has been applied to a variety of problem domains, including security, network protocols, mutual exclusion algorithms, VLSI chip designs and chemical manufacturing systems. In recent years, CPN has been used in the areas of fault management and security system. It is very well suited for designing IDS because it can describe clearly the complicated and complex interaction, classification and correlation of activities of IDS. Moreover, CPN design of IDS provides an efficient way to verify IDS design. Since IDS is difficult to test because intrusion activities are complicated to simulate, this is very beneficial. The Design/CPN tool from the University of Aarhus, Denmark, is used in our project to draw and analyze CPN representations of the IDS system design, and simulate attacks to verify the CPN representation of the IDS design. This tool allows us to both save and load the CPN diagrams from eXtensible Markup

Language (XML) files. Our compiler uses the XML representation of the CPN diagrams as input to the automatic code generation of our agent system.

2.4. SFT to CPN translation

SFT can model intrusions and help in requirement analysis for an IDS. However, it is not a good model for IDS design. The reason is that SFT can only describe the events and sequences of events that cause the intrusions in a system, and it does not describe the procedure to detect and correlate these events. To model an IDS design, an execution model like CPN is needed.

CPNs are very powerful in modeling IDS, but, they are complicated. A designer must not only have adequate knowledge in the intrusion domain, but also have a good command of CPN. Tremendous effort is needed to manually develop a CPN for an IDS design.

Thus, we propose a way to map from an SFT description of attacks to CPN design templates for IDS (Wang et al., submitted for publication). In this paper, we present the MAIDS compiler which makes the translation from the CPN specification of IDS to MAIDS software agents.

3. MAIDS agent system design

The unique feature of the MAIDS project is that it is developed using a disciplined requirements engineering process. The entire process comprises of primarily three steps.

1. A high-level description of intrusions is created using a SFT.
2. This description is then automatically translated into a CPN which serves as the design specification for IDS. The CPN design specification for the IDS can be optimized and verified using formal tools such as Design/CPN (Denmark CPN Group at University of Aarhus, 2003).
3. The CPN is then translated into the actual implementation of IDS software agents.

The MAIDS development hierarchy is given in Fig. 1. In the following we present the basis for the translation from the second stage to the third stage.

The MAIDS software agent system closely follows the system design as encapsulated by the CPN representation. Leaf places correspond to data source agents. Each leaf place in the original CPN is created at each host throughout the network, and is responsible for creating tokens and holding them until they are picked up by other mobile agents. Data source agents communicate with a local database that is being populated by some local data sources, such as an auditing server and other IDSs.

A leaf transition has leaf places among its inputs and corresponds to a mobile agent. These mobile agents travel through the computer network, pick up tokens and correlate events at every site that they visit. Migration of the leaf transitions is horizontal (circulating among the monitored hosts) rather than radial (making repeated trips to and from the analyst's location).

The root place in the CPN corresponds to the console of the administrator who is monitoring the system. The internal places correspond to stationary agents that hold tokens, and the internal transitions are similar to leaf transitions except for they are stationary.

At each monitored host there is an agent server, a local database, and an assortment of local data sources. At the analyst's location there is another agent server to serve as the creating point of all agents and a console application that serves administrative role for all agents. Fig. 2 illustrates the overall architecture of the MAIDS agent system.

4. Compiler design

The design of the compiler has the following goals:

Correctness: It should preserve the correctness of the CPN in the translation;
Minimal manual intervention: The whole idea behind having the tool is to enable system administrators to use MAIDS without knowing the internals of the system. So it should require minimal manual intervention from the users in the translation;
Generality: It should not restrict the ability of the CPN to represent a wide variety of IDS;
Flexibility: Though the Design/CPN tool (Denmark CPN Group at University of Aarhus, 2003) is the most popular way to represent CPN, the tool should not constrain the users to use it. If the user decides to use some other representation of CPN, the changes that have to be made to our automated tool to handle this should be as painless as possible;

The compiler assumes that the IDS design as specified by the CPN is correct. If there are flaws in the CPN they will get passed on to the agent code. The correctness of the CPN can be checked and analyzed using the Design/CPN tool by simulation (Denmark CPN Group at University of Aarhus, 2003).

4.1. Translation of three basic elements

A CPN diagram is composed of three basic elements - places, transitions and arcs. Each of these elements corresponds to a certain class in the MAIDS agent system. We provides the following CPN elements to MAIDS class mapping based on the description in Section 3.

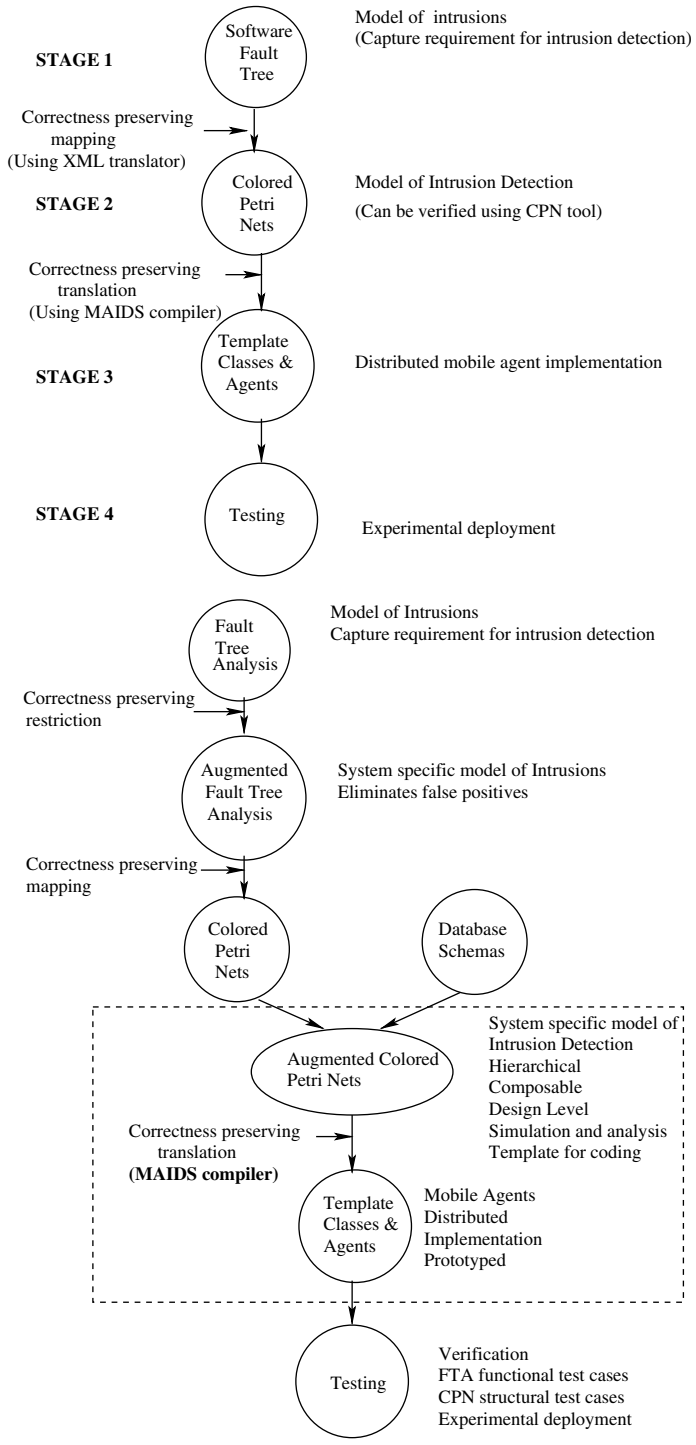

Fig. 1. IDS development process.

4.1.1. Places

Places, which have no incoming arcs, are called Leaf places (DataPlace class in MAIDS). All other places are called Inner places. Leaf places are equivalent to Data Source agents, which monitor the database and generate tokens when some events happen. To create a data source agent the following information has to be extracted from a leaf place in the CPN diagram.

1. Name.
2. A short description.
3. Database information (name of table and table field in database monitored by this place).
4. Tokens generated at this place. Detailed information about the tokens is extracted from the arcs.

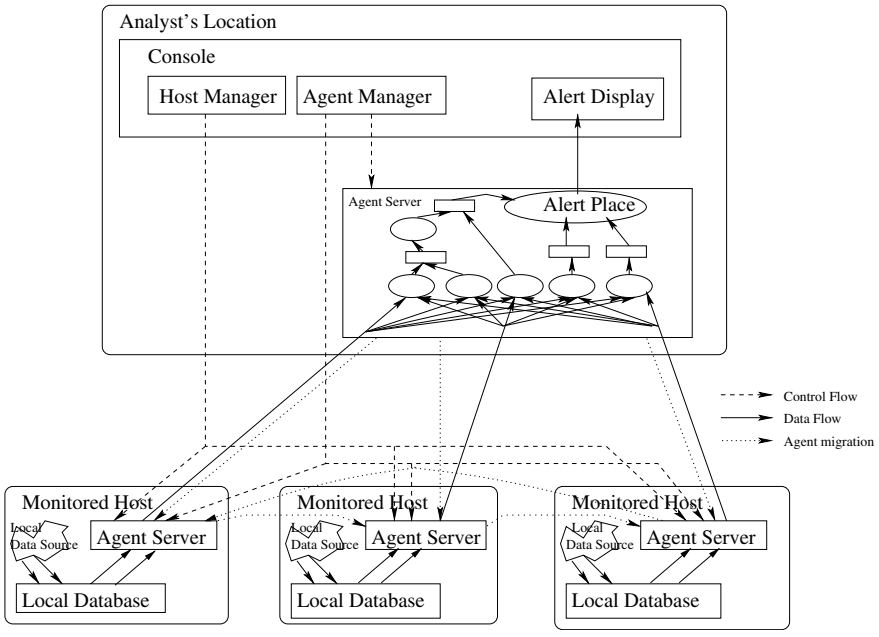

Fig. 2. MAIDS architecture.

For each inner place, there's no agent generated for it and it serves as a container for tokens. To create an instantiation of the inner place class with a unique label we need to extract the name for an inner place.

4.1.2. Transitions

Transitions that have at least one Leaf place as a source place are called Leaf transitions. All other transitions are Inner transitions. Leaf transitions are equivalent to Mobile Transition agents (MobileTransition class), and Inner transitions are equivalent to Stationary Transition agents (StationaryTransition class) in MAIDS. Both Mobile Transition and Stationary Transition agents have the same code. The only difference between them is their mobility which is hidden in their superclass code. So for all transitions in the CPN, the following information has to be extracted.

1. Name of the transition.
2. Names of source places.
3. Names of incoming arcs.
4. The formula used to unify tokens in the transition (guard formula).
5. Tokens generated by the transition. Detailed information about these tokens is extracted from the arcs.

4.1.3. Arcs

For arcs and tokens in CPN, we use a Token class to represent it. The following information needs to be extracted from each arc.

1. Name of arc.
2. Its source place or transition.
3. Its destination place or transition.
4. The tags of the data it is carrying.
5. For arcs that originate from leaf places we also need to extract database information. This includes names of the fields in the database from which the token data has to be obtained, and the value of the field in the database, which would result in the generation of a token.

4.2. Algorithms for the translation

At a high level, the compiler design can be illustrated by Algorithm 1.

Algorithm 1. (Compiler)

Open CPN file
for each element in the CPN file **do**
  Determine the type (leaf or inner place, leaf or inner transition, arc) of the element
  Extract information about the element
  Use the type and the mapping given above to decide what MAIDS class to create
  Use the extracted information to create this MAIDS class file
end for
Close CPN file

The MAIDS compiler comprises two distinct stages. A set of objects, which holds information about the three CPN elements mentioned above, serves as the interface between these two stages. The first

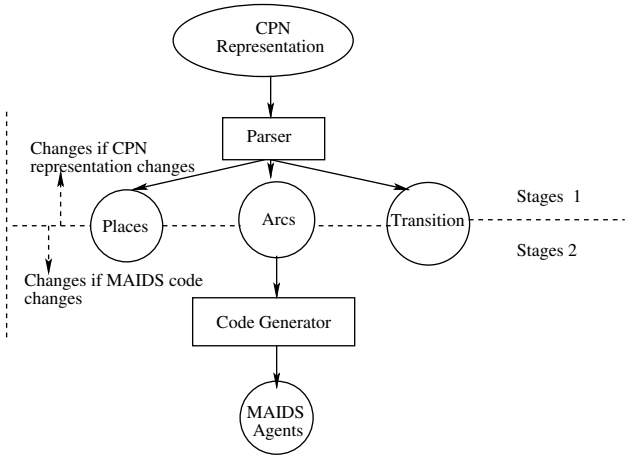

Fig. 3. Design of the MAIDS compiler.

stage parses the CPN representation, extracts information about each CPN element and populates these objects. The second stage uses the data stored in these objects to generate MAIDS agent code. The flexibility goal of the compiler can be achieved, as a new CPN representation would involve changing only the first stage.

The steps involved in the working of the compiler is as Fig. 3, the pseudo code for this two-stage design is as Algorithm 2.

Algorithm 2. (Two-stage compiler)

Open CPN file
for each element in the CPN file **do**
  Determine the type of the element
  Extract information about the element
  Determine the object based on the type obtained
  Put the extracted information in the object
end for
Close CPN file
for each object in the interface object set **do**
  Determine the type of the element to which it corresponds
  Get the data from this object
  Use the type obtained above to determine the MAIDS class to be created
  Use the data to create this MAIDS class
end for

4.3. Evaluation of design

The compiler design has achieved the design goals:

Correctness: The compiler strictly follows the CPN element to MAIDS agent translation outlined in Section 3.

Minimal manual intervention: No manual intervention is needed in the transition procedure.

Generality: The design does not make any assumptions that are specific to a particular attack. So the compiler can generate software agent code to detect any attack where the intrusion detection can be modeled using a CPN.

Flexibility: The design makes a clear distinction between the parsing and the code generation stages of the compiler. Moreover it defines a clear interface between these two stages through the Place, Transition and Token objects. This ensures that if the CPN representation changes, then only the parsing stage of the compiler needs to be changed.

5. Implementation

The current MAIDS implementation uses the Voyager agent platform, version 4.5, from Recursive Software (2003). At each monitored host and console machine a Voyager server has to be running. In this section, we first introduce the major classes we used in the compiler.

5.1. MAIDS agent classes

The hierarchy of the agent classes is given in Fig. 4.

Leaf (data source) places: The responsibility of a data place is to generate fresh tokens from local event databases. It must implement a work( ) method, taking no arguments and returning a TokenBag. This method is called periodically within the DataPlace superclass code.

Leaf (mobile) transitions: This embodies both the topology and behavior of the CPN. This must implement the following methods

```
java.lang.String[ ] sources( );
java.lang.String[ ] tokenSpec( );
Token[ ] unify (Token[ ] sourceTokens);
```

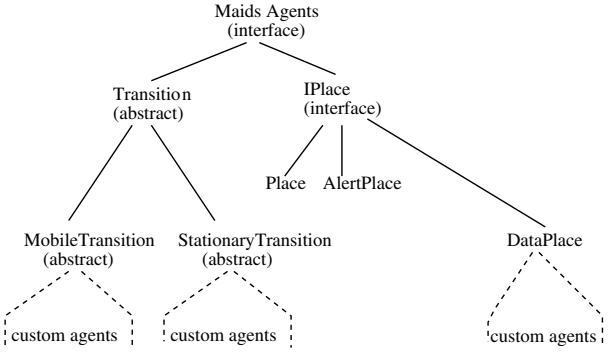

Fig. 4. The MAIDS class hierarchy.

Of these methods, the only non-trivial method is 'unify'. This method decides whether the tokens passed to it should be unified. If so, it returns a new array of tokens; else it returns null. As for the other two methods, sources( ) returns an array of the labels of the places that have arcs to this source, and tokenSpec( ) returns an array of token colors and implicitly determines what kind of input the unify( ) method will see.

Internal places: No special agent needs to be written for this as it only serves as a container.

Internal transitions: These are very similar to Leaf transitions with the only difference that they need not be mobile.

Root (console) place: There is no need to instantiate the root place. This is automatically done by the MAIDS console which implements the IPlace interface and identifies itself in the CPN with the alert label.

5.2. Compiler class structure

The following is a description of the major classes used by the MAIDS Compiler. Fig. 5 illustrates the MAIDS compiler implementation.

PlaceToken: This class stores all the information that is required to create a Token object in the MAIDS DataPlace class. This includes token name, description and data.

PlaceInfo: This class stores all the information that is required to create a subclass of the MAIDS DataPlace class. This includes place name, tokens and database information for that place.

DBInfo: This is a simple class that serves as a container to hold database information (name of database table, name of fields in this table). The PlaceInfo class uses this class to create a subclass of the DataPlace class.

TransitionToken: This class stores all the information that is needed to create a Token object in the MAIDS StationaryTransition or MobileTransition class. This includes token name, data, destination and urgency.

TransitionInfo: This class stores all the information required to create a subclass of the MAIDS StationaryTransition or MobileTransition class. This includes transition name, type, sources, unifying formula and tokens.

Parse: This is the abstract class that all CPN parser classes have to extend. All subclasses of this class have to implements a main( ) function which takes the name of the CPN file as input, and returns the following vectors:

v[0]—A vector containing all the PlaceInfo objects that are derived from the leaf places in the CPN representation.

v[1]—A vector containing all the TransitionInfo objects that are derived from the transitions in the CPN file.

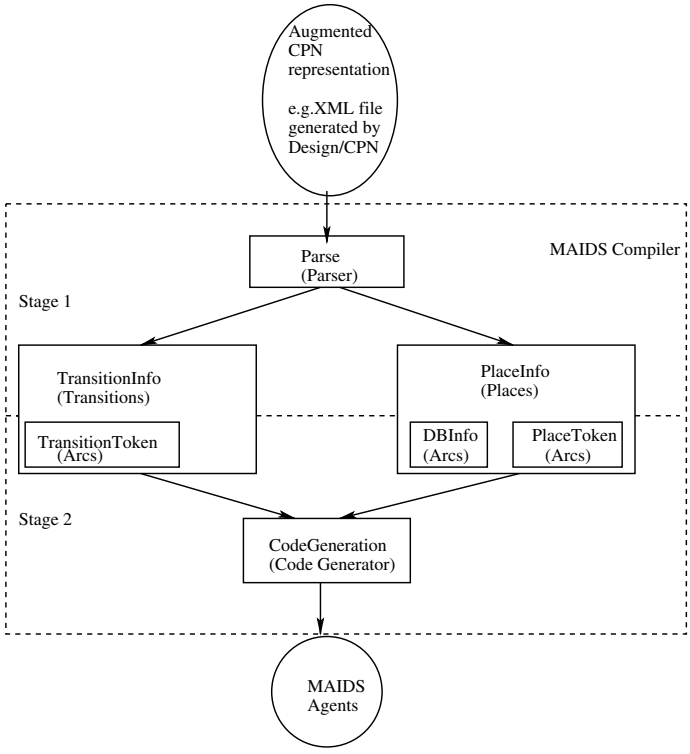

Fig. 5. Implementation of the MAIDS compiler.

v[2]—A vector containing names of all the Inner places in the CPN file.

CodeGen: This is the class that does the actual code generation. It takes the Vector returned by the main( ) method of the Parse class as its input. For each PlaceInfo object it creates a class which extends the MAIDS DataPlace class. For each TransitionInfo object it checks the type. If the type is 'Mobile', it creates a class that extends the MAIDS MobileTransition class. If the type is 'Stationary', it extends the MAIDS StationaryTransition class.

MAIDS provides a way to start the system from a specifications file 'spec.txt'. This file contains the names of the agents as well as Voyager host information (Slagell, 2001). The CodeGen class creates a 'spec.txt' file that contains the names of all the classes it created.

5.3. Database information

Data Source agents need to interact with the databases to get the information on monitored system to produce tokens. To create the Data Source agents, information about the names of the tables and fields being monitored need to be extracted from the CPN. These tables in the local database must be populated by some data sources, such as auditing tools and other IDSs.

5.4. XML representation of CPN using design/CPN tool

The Design/CPN tool (Denmark CPN Group at University of Aarhus, 2003) (versions 4.0.4 and higher) allows the user to save all CPNs as XML files. This tool is a very widely used tool to represent CPN. The XML representation generated by this tool is used as input to the compiler.

The CPN parser class, which extends the Parse class, can be used to parse representations of this class, assuming that the XML uses 'cpnet.dtd' as the style sheet. This class assumes that the CPN diagram drawn using this tool satisfies the following minimum requirements.

1. The CPN diagram should be well-formed, i.e., there should be no arc that connects directly a place with a place or a transition with a transition.
2. All transitions, places and arcs should have names associated with them.
3. Every place should also have a color associated with it. All tokens coming into this place or originating from this place should be of this same color.
4. Every arc should have a token description associated with it.

If the style sheet changes, this class will need to be changed.

5.5. Incorporating a new CPN representation

A new CPN representation can be incorporated by the steps outlined below.

1. Implement the Parser class, which extends the Parse class. This class should implement the main( ) method of the Parse class as mentioned previously.
2. Examine the constructors of PlaceInfo, TransitionInfo, DBInfo, TransitionToken and PlaceToken classes to fill in the correct information into the corresponding objects.
3. Recompile and run the compiler on the new representation.

The end user can easily incorporate any representation of the IDS. The system does not need to be reset when rules are updated. The new agents can be generated and added to the system dynamically.

When some nodes are down and some agents are lost, there is no impact on the functionality of other nodes and other agents. When the nodes go up, new agents can be launched again without impacting the current agents.

In the following section, we demonstrate two examples of automatic generation of intrusion detection agents from the CPN intrusion detection design templates.

6. Experiments and results

In this section, we will describe two attacks in the experimental scenario, namely, the FTP bounce attack and the Trinoo DDoS attack.

6.1. FTP bounce attack

This attack exploits a weakness in certain FTP daemons and a trust relationship between two hosts. A detailed explanation of this attack is provided in (CERT Coordination Center, 2003b). The basic steps involved are:

1. Create a malicious file containing a valid remote shell (RSH) message.
2. Identify a vulnerable FTP server (relay host) and a host which trusts the relay host and is running a RSH daemon (victim).
3. Upload the file to the relay host.
4. Redirect the output of the FTP server to the RSH port on victim using the PORT command.
5. Download the file directly to the RSH port of the victim.
6. The victim accepts the file as a command to be run as root user.

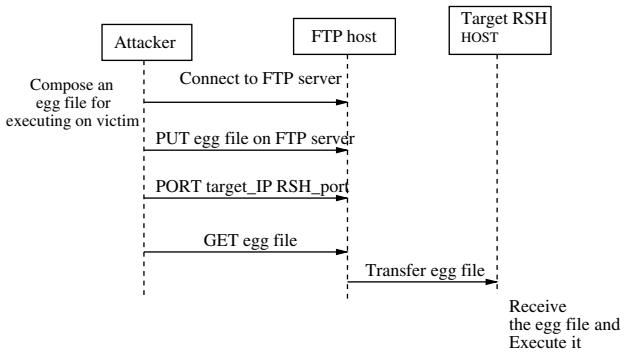

Fig. 6. Diagraph for FTP bounce attack.

FTP bounce attack is a good candidate for testing MAIDS as it is a distributed attack involving three machines (attacker, relay and victim) as show in Fig. 6.

The CPN diagram describing the intrusion detector for detecting this attack, as displayed by the Design/CPN tool, is given in Fig. 8 (the text representation of CPN for FTP bounce attack is in Appendix A). As can be seen from the CPN diagrams, there are five events which constitute positive identification of the FTP bounce attack. Four of these—FTP PORT request, FTP PORT success, FTP retrieve request and FTP retrieve success—take place at the relay host and the other one—RSH connection from FTP—takes place at the victim host.

The CPN diagram for FTP bounce attack depicted in Fig. 8 was saved as XML files using the Design/CPN tool and input into the MAIDS compiler. The following files were created:

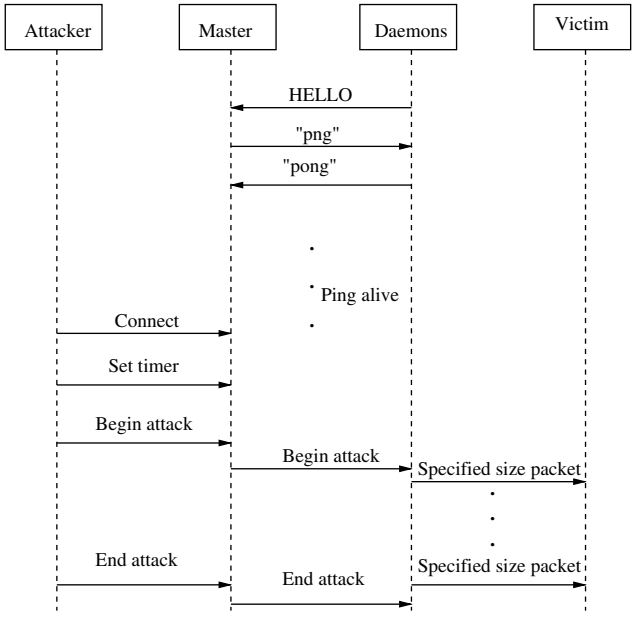

Fig. 7. Diagraph for TRINOO DDOS attack.

Places: NetworkMonitorFTP.java and NetworkMonitorTCPConnections.java
Transitions: FTP_PORTFTP_PORT_OK.java, FTP_RETR.java, FTP_RSH_CONN.java and FTP_BOUNCE_ATTACK.java
spec.txt: This is a specification file used by MAIDS to start the agent system to detect FTP bounce attack. The agent names for detection of the FTP bounce attack are listed. The intrusion detection agents can also be loaded after MAIDS starts.

An intrusion script was used to launch the attack from another host outside our local network. Two monitored Linux hosts were involved, one as relay and the other as target. The relay host was running a vulnerable ftp server, which enables PORT commands regardless of source or destination.

The frequency of events that MAIDS can handle depends on the speed of data preprocessing. In the MAIDS system, the data is supposed to have been preprocessed. The data that is used by the MAIDS system is pre-aggregated by software such as SNORT or Tripwire.

The compiler does not deal with optimization. The optimization is done in CPN design phase. The CPNs that represent the IDS design are optimized in Design/CPN software, and then compiled into agent code using the compiler.

The performance of the MAIDS compiler generated agents are compared with the performance of the hand coded agents we used in our earlier research. For the first experiment (low-traffic attack) an attack script was used to generate 2 ftp attacks interspersed between 8 normal GET/PUT sequences. For second experiment (high-traffic attack) an attack script was used to generate 2 ftp attacks interspersed between 48 normal GET/PUT sequences. Both the generated and the hand-coded agents were able to detect the FTP bounce attacks without giving any false positives or false negatives.

Their relative performances from an efficiency viewpoint, as summarized in Table 1 show that there is no appreciable difference in the detection times between hand coded agents and compiler generated agents.

6.2. TRINOO DDoS attack

DDoS attack is a massive coordinated attack and normally involves hundreds or even thousands of machines that simultaneously attack the victim. In DDoS attack, there are one or more control master programs and many daemons controlled by the masters. When masters issue attack commands to daemons, the daemons launch simultaneous attacks, such as flooding the victim (CERT Coordination Center, 2003c). This will disrupt normal operation of the victim or even bring down the victim.

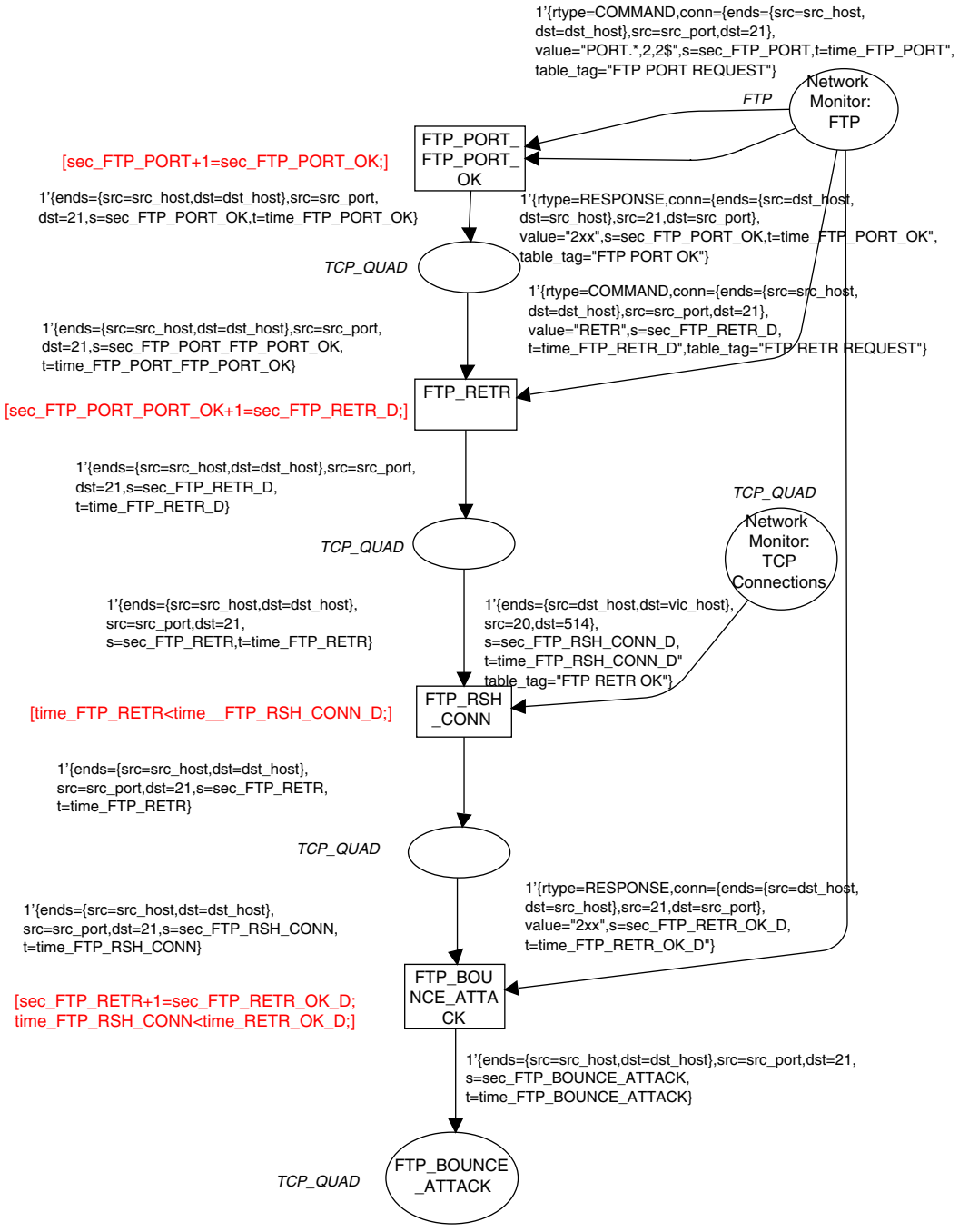

Fig. 8. CPN for FTP bounce attack.

Table 1
Detection times for the basic FTP bounce attack (in seconds)

	Low-traffic attack	High-traffic attack
Hand coded agents	11	15
Compiler generated agents	13	16

The Trinoo DDoS attack (Dittrich, 2003) is used as test case in our experiment. Trinoo DDoS attacks normally consist of two steps. First, a Trinoo network is created, composed of one or several masters and many daemons controlled by the masters. Second, the attack is launched by issuing an attack command to the master servers from the attacker. After getting the attack command from the attacker, the masters will command all the daemons to send packets to the victim machine.

Communication from Trinoo masters to daemons is via 27444/udp, and from Trinoo daemons to masters is via 31335/udp by default. When a daemon starts, it

initially sends a "*HELLO*" command to the master. Masters maintain a list of active daemons controlled by them and use a keep-alive procedure to find out which daemons are alive. When a Trinoo master sends a "png" command to a daemon it controlled, the daemon need to reply to the master by sending string "pong" to indicate it is alive.

An attacker remotely controls Trinoo masters via TCP connections to port 27665/tcp (by default) on the masters' host. To start up a DDoS attack, an attacker need to connect to this port and issue commands. The attacker can issue a "mtimer N_seconds" command to set the attack time and then issue "dos victim_ip" command to begin the DDoS attack. When masters receive this DoS attack command, they issue "aaa password victim_ip" command to all daemons they control to launch DoS attacks to the victim machine.

The detailed explanation of the Trinoo attack can be found in (Dittrich, 2003). Fig. 7 shows the communications in TRINOO attack. Fig. 9 is the CPN for detecting TRINOO DDoS attack.

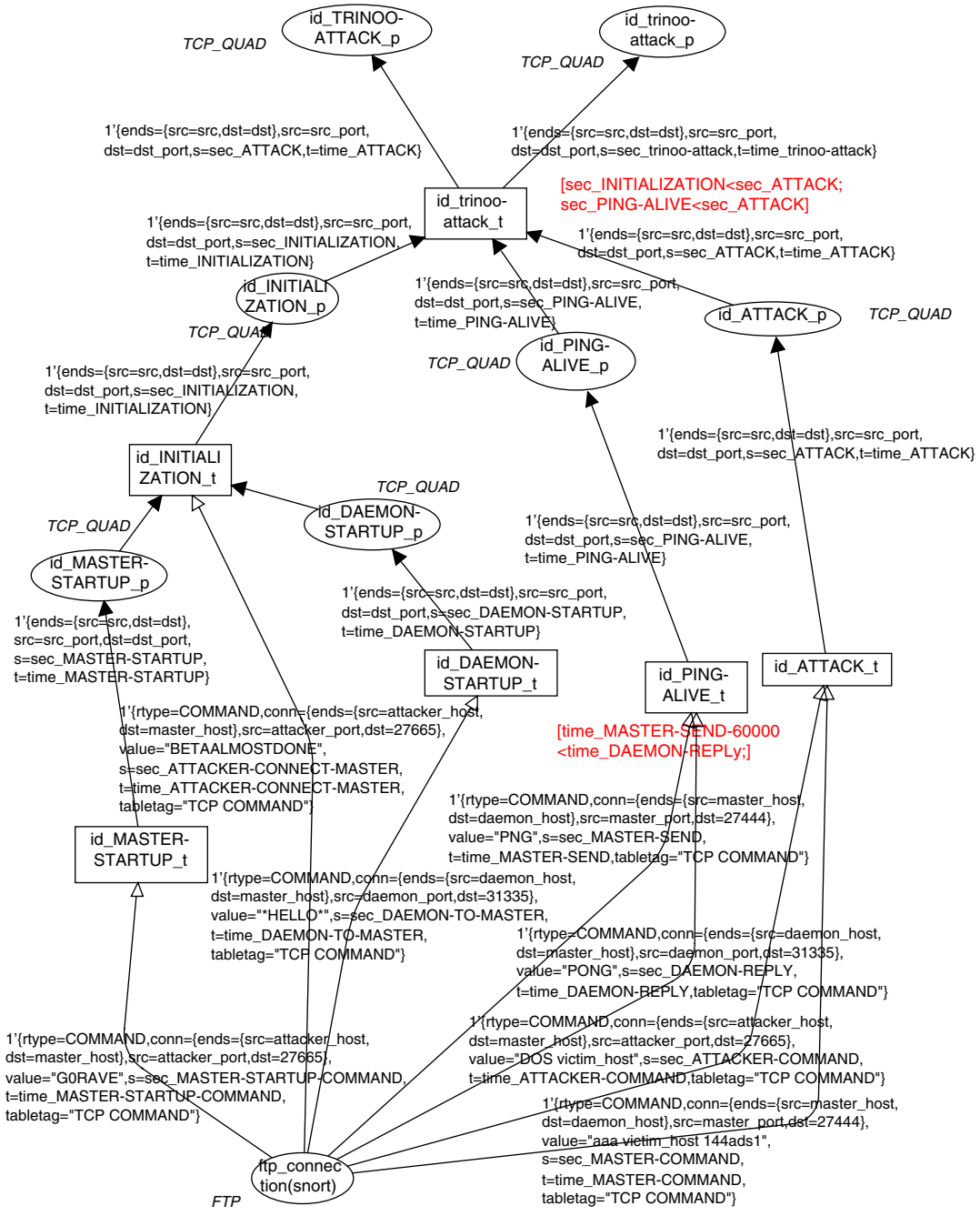

Fig. 9. CPN for TRINOO DDOS attack.

The CPN diagram for TRINOO DDoS attack depicted in Fig. 9 was saved as XML files using the Design/CPN tool and input into the MAIDS compiler. The following files were created:

Places: NetworkMonitorTCPConnections.java
Transitions id_MASTER-STARTUP_t.java id_DEAMON-STARTUP_t.java id_PING-ALIVE_t.java id_ATTACK_t.java id_INITIALIZATION_t.java id_trinoo-attack_t.java
spec.txt: This is a specification file used by MAIDS to start the agent system to detect DDoS attack.

We set up an experiment to test the intrusion detection software agents automatically generated from the CPN, using the Trinoo DDoS attack described above. The generated software agents were able to detect the attack without giving any false positives or false negatives.

The performance of the MAIDS compiler generated agents was compared with the performance of the hand coded agents we used in our earlier research. The average detection time was 25 seconds using hand coded agents, and 22 seconds using compiler generated agents.

7. Conclusions and future work

The main advantages of MAIDS is the use of a formal method to assure that the IDS satisfies a set of high-level requirements (SFT specification of intrusions). Formal method and formal tools are used in the development of the IDS to reduce design and implementation errors. CPN representations of IDS design template, which can be recognized and displayed by Design/CPN tool, are used as input to a compiler to generate software intrusion detection agents. Design/CPN allows the correctness of the IDS design to be checked before moving into the implementation stage.

A unique feature of MAIDS is the automated tools to generate intrusion detection agents given a high level description of intrusions. The compiler presented in this paper is an important step toward automated tools to generate intrusion detection agents. The automatic generation of software intrusion detection agents from description of intrusions is beneficial because it not only prevents human coding errors from compromising the correctness of the IDS system design, but also provides a convenient way for end users to generate and launch new intrusion detection agents when new intrusion types are found. The simplicity of updating MAIDS to detect new intrusions is important since new intrusion types emerge every day. Some IDSs like "snort" provide configuration files which can be updated to include new intrusion signatures (Caswell and Roesch, 2003). However, the rules in their configuration files can only describe simple signatures, and most complicated and coordinated intrusions cannot be described precisely and conveniently. Our tool provides an easy way of describing sophisticated intrusions using a SFT model and generating the intrusion detection agents to detect the new intrusions.

Currently, we use simple token unification based on timestamps, sequence numbers and machine IP addresses. Nowadays, attackers often move around many machines before a real attack is launched or use fake IP addresses in attacking, which makes the trace back very difficult. Now we are looking at how to unify tokens based on other information such as user identifiers so the correlation of events among machines can be more effective and trace back can be easier.

Appendix A. Text representation of CPN for FTP bounce attack

PLACE:

Name: FTPB_relay_DS
Header: ftp monitor
Token: FTP_PORT, FTP PORT REQUEST, ftp PORT requested, cid | sequence, *timestamp | time: FTP_PORT_OK, FTP PORT SUCCESS, ftp PORT succeeded, cid | sequence, *timestamp | time: FTP_RETR, FTP RETR REQUEST, ftp RETR requested, cid | sequence, *timestamp | time: FTP_RETR_OK, FTP RETR SUCCESS, ftp RETR succeeded, cid | sequence, *timestamp | time
DB: org.gjt.mm.mysql.Driver, mysql, event, signature

PLACE:

Name: FTPB_victim_DS
Header: tcp monitor
Token: FTP_RSH_CONN, RSH CONNECT FROM FTP, rsh connection from ftp, cid—sequence, *timestamp—time DB: org.gjt.mm.mysql.Driver, mysql, event, signature

TRANSITION:

Name: FTPB_MT1
Source: *FTPB_relay_DS
InToken: FTP_PORT, FTP_PORT_OK
Formula: I_1_sequence + 1 = I_2_sequence
OutToken: FTP_PORT & FTP_PORT_OK,seq2, I_2_sequence,FTPB_Place1

PLACE:

Name: FTPB_Place1

TRANSITION:

Name: FTPB_MT2

Source: *FTPB_relay_DS, FTPB_Place1
InToken: FTP_PORT & FTP_PORT_OK, FTP_RETR
Formula: $I\_1\_seq2 + 1 = I\_2\_sequence$
OutToken: FTP_RETR,time3 seq3,L_2_time I_2_sequence,FTPB_Place2

PLACE:

Name: FTPB_Place2

TRANSITION:

Name: FTPB_MT3
Source: *FTPB_victim_DS, FTPB_Place2
InToken: FTP_RETR, FTP_RSH_CONN
Formula: $L\_1\_time3 - 300000 < L\_2\_time$
OutToken: FTP_RSH,seq3 time4,I_1_seq3 L_2_time, FTPB_Place3

PLACE:

Name: FTPB_Place3

TRANSITION:

Name: FTPB_MT4
Source: *FTPB_relay_DS, FTPB_Place3
InToken: FTP_RSH, FTP_RETR_OK
Formula: $I\_1\_seq3 + 1 = I\_2\_sequence$ AND $L\_1\_time4 - 300000 < L\_2\_time$
OutToken: FTP_BOUNCE_ATTACK,NULL,NULL, alert,9

References

Caswell, B., Roesch, M., 2003. Snort—The Open Source Network Intrusion Detection System. Online, date accessed: October 2003. Available from: <http://www.snort.org>.

CERT Coordination Center, 2003a. FTP bounce. Online, date accessed: October 2003. Available from: <http://www.cert.org/advisories/CA-1997-27.html>.

CERT Coordination Center, 2003b. Problems with the FTP PORT Command, 2003. Available from: <http://www.cert.org/tech_tips/ftp_port_attacks.html>.

CERT Coordination Center, 2003c. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks. Online, date accessed: October 2003. Available from: <http://www.cert.org/advisories/CA-1996-21.html>.

Denmark CPN Group at University of Aarhus, 2003. Design/CPN online. Online, date accessed: October 2003. Available from: <http://www.daimi.au.dk/designCPN/>.

Dittrich, D., 2003. The DoS Project's "Trinoo" Distributed Denial of Service Attack Tool. Online, date accessed: October 2003. Available from: <http://staff.washington.edu/dittrich/misc/trinoo.analysis.

Helmer, G., Wong, J., Slagell, M., Honavar, V., Miller, L., Lutz, R. 2002. A Software Fault Tree approach to requirements analysis of an intrusion detection system. Requirements Engineering Journal, 7 (4), 207–220.

Inc. Recursive Software. Voyager, 2003. Available from: <http://www.recursionsw.com/products/voyager/voyager.asp>.

Information Security Lab, 2003. MAIDS Intrusion Detection System Project. Online, date accessed: October 2003. Available from: <http://latte.cs.iastate.edu/Research/Intrusion/index.html>.

Intrusion Detection Hotlist, 2003. Online, date accessed: October 2003. Available from: <http://www.cerias.purdue.edu/coast/ids/>.

Jensen, K., 1992. Basic Concepts, Analysis Methods and Practical Use, vol. 1. Springer-Verlag, Germany/Heidelberg, Germany/London.

Lee, W., Stolfo, S., Mok. K., 1999. A data mining framework for building intrusion detection models. In: IEEE Symposium on Security and Privacy, pp. 120–132.

Porras, P.A., Neumann, P.G., 1997. EMERALD: event monitoring enabling responses to anomalous live disturbances. In: Proceedings of the 20th National Information Systems Security Conference, Baltimore, Maryland, October 1997, pp. 353–365.

Slagell, M., 2001. The Design and Implementation of MAIDS (Mobile Agents for Intrusion Detection System). Master's thesis, Iowa State University, May 2001.

Snapp, S., Brentano, J., Dias, G., Goan, T., Heberlein, L., Ho, C., Levitt, K., Mukherjee, B., 1991. A system for distributed intrusion detection. In: COMPCON Spring'91, Digest of Papers, San Francisco, CA, February 1991, pp. 170–176.

Spafford, E.H., Zamboni, D., 2000. Intrusion detection using autonomous agents. Computer Networks 34 (4), pp. 547–570.

Wang, Y., Wong, J., Helmer, G., Miller, L., Honavar, V., Lutz. R., Towards a formal approach to the design and implementation of intrusion detection systems. International Journal of System Science, submitted for publication.