Software Fault Tree and Colored Petri Net Based Specification, Design and Implementation of Agent-Based Intrusion Detection Systems

Guy Helmer[*], Johnny Wong,[†] Mark Slagell,[†] Vasant Honavar,[†] Les Miller,[†]
Yanxin Wang, Xia Wang, Natalia Stakhanova

Department of Computer Science
Iowa State University
{ghelmer,wong,slagell,honavar,lmiller,wangyx,jxiawang,ndubrov}@cs.iastate.edu

March 16, 2006

Abstract

The integration of Software Fault Tree (SFT) which describes intrusions and Colored Petri Nets (CPNs) which specifies design, is examined for an Intrusion Detection System (IDS). The IDS under development is a collection of mobile agents that detect, classify, and correlate system and network activities. Software Fault Trees (SFTs), augmented with nodes that describe trust, temporal, and contextual relationships, are used to describe intrusions. CPNs for intrusion detection are built using CPN templates created from the augmented SFTs. Hierarchical CPNs are created to detect critical stages of intrusions. The agent-based implementation of the IDS is then constructed from the CPNs. Examples of intrusions and descriptions of the prototype implementation are used to demonstrate how the CPN approach has been used in development of the IDS.

The main contribution of this paper is an approach to systematic specification, design, and implementation of an IDS. Innovations include (1) using stages of intrusions to structure the specification and design of the IDS, (2) augmentation of SFT with trust, temporal, and contextual nodes to model intrusions, (3) algorithmic construction of CPNs from augmented SFT, and (4) generation of mobile agents from CPNs.

1 Introduction

A secure computer system provides guarantees regarding the confidentiality, integrity, and availability of its objects (such as data, processes, or services). However, systems generally contain design and implementation flaws that result in security vulnerabilities. An intrusion takes place when an attacker or group of attackers exploit security vulnerabilities and thus violate the confidentiality, integrity, or availability guarantees of a system or a network. Intrusion detection systems (IDSs) detect some set of intrusions and execute some predetermined action when an intrusion is detected.

IDSs use audit information obtained from host systems and networks to determine whether violations of a system's security policy are occurring or have occurred [4]. Our Multi-Agents Intrusion Detection System (MAIDS) [21, 19, 20] uses mobile agents [9] in a distributed system to obtain audit data, correlate events, and

[*] G. Helmer is with Palisade Systems, Inc. His research was funded in part by the Department of Defense, the Boeing Company in the form of the Boeing Dissertation Fellowship, and the Graduate College of Iowa State University.

[†]Funded in part by the Department of Defense. Corresponding author: Johnny Wong, 202 Atanasoff Hall, Department of Computer Science, Iowa State University, Ames, Iowa 50011

discover intrusions. The MAIDS system consists of (1) stationary data cleaning agents that obtain information from system logs, audit data, and operational statistics and convert the information into a common format, (2) low level agents that monitor and classify ongoing activities, classify events, and pass on their information to mediators, and (3) data mining [10] agents that use machine learning to acquire predictive rules for intrusion detection from system logs and audit data.

One of the challenges in designing an IDS involves defining exactly what data elements should be correlated to determine whether an intrusion is taking place in a distributed environment. It is also difficult to determine what data elements are necessary to discover intrusions. A model of intrusion detection is essential to describe how the data should flow through the system, determine whether the system would be able to detect intrusions, and suggest points at which countermeasures could be implemented.

Against this background, the paper presents a theoretical framework for modeling the operation of intrusion detection systems such as MAIDS. We use Software Fault Trees (SFTs) to define intrusions and develop the requirements model for the IDS. The SFT models of intrusions are used to create Colored Petri Net (CPN) designs for the detectors in the IDS. The CPN detection model is then mapped into a set of software mobile agents that form the distributed intrusion detection system. Finally, the SFT models provide test cases for the implementation.

The SFT analysis (SFTA) approach applies safety engineering techniques to the intrusion detection domain for developing IDS requirements. Each part of these development processes — SFTA, CPNs, and software agent implementation — is distinct, and each stage in the development process must correctly carry over the details of the previous stages. The constructive approach helps ensure the correctness of the design with respect to requirements and correctness of the implementation with respect to the design.

We present the process for developing a CPN design for the IDS using a requirement specification based on a SFTA of intrusions, and we show the procedure for creating an implementation of a distributed agent-based IDS from the CPN design [61]. These two procedures ensure that the design satisfies the requirements and that the implementation matches the design.

The rest of this paper is organized as follows: Section 2 introduces temporal organization of stages of intrusions and presents the intrusions examined in our research. Section 3 discusses SFTs as applied to modeling intrusions and the augmentations needed to describe intrusions. Section 4 introduces CPNs and defines the translation from augmented SFTs to CPNs. Section 5 defines the translation of CPNs to software mobile agents. Section 6 presents the intrusion scenarios evaluated in our system. Section 7 relates our modeling solution to other graph-based intrusion detection models. Section 8 presents the conclusions and contributions of this work, discusses the generalization of the IDS design to intrusions other than those presented in this paper, and describes future work.

2 Modeling Intrusion Detection

Our goal is to develop a software model for precisely describing a broad class of intrusions as well as the process of detecting such intrusions. Any formalism used to define the intrusions has to be relatively easy to use and at the same time be rich enough to describe both single host and distributed attacks. Software Fault Trees (SFTs) provide the desired features. When combined with Software Fault Tree Analysis (SFTA) they provide an effective means for defining intrusions in a way that exposes the critical aspects of determining an intrusion.

However, SFTs have several limitations with regard to their ability to modeling intrusions. For example, in their simplest form, they cannot capture temporal relationships between events. Hence, we extend the SFT formalism to obtain augmented SFTs. By extending SFTs to *augmented* SFTs, SFTs with additional system information, it is possible to create a rigorous process that is capable of capturing intrusions.

But, even the augmented SFTs do not describe intrusions at a level of detail needed to automate the generation of software agents that implement an IDS. To bridge this gap, we have developed a rigorous approach to convert the augmented SFTs to Colored Petri Nets (CPNs). While the augmented SFTs provide a rigorous definition of the intrusions, the CPNs provide a rigorous definition of the process of detecting the

intrusion. Then a conversion of CPNs to software mobile agents can be performed.

Each augmented SFT is seen as the specification of an individual intrusion. The set of augmented SFTs is the model of intrusions that the IDS is able to detect and/or determine. The set of CPNs that can be generated from the set of augmented SFTs is the intrusion detection model.

2.1 Temporal Stages of Intrusions

Each successful intrusion can vary greatly from other intrusions. In addition, analysis of complete intrusions is quite difficult. Therefore, a reasonable approach to intrusion analysis is to divide attack into stages that achieve intermediate goals of the attacker and develop intrusion detection components that identify each of the stages. Generally, the following stages can be distinguished in intrusion analysis: Reconnaissance, Vulnerability Identification, Penetration, Control, Embedding, Data Extraction & Modification, and Attack Relay [19].

We use these seven stages to analyze the intrusion example discussed in this paper and reduce the complexity of each SFT. The CPNs examined in the paper generally correspond to the first three stages: reconnaissance, vulnerability identification, and penetration, as essential stages of intrusion [19].

2.2 Intrusion Example

We use the FTP bounce attack throughout the paper to illustrate our approach to specification and implementation of IDS. The example was chosen based on the fact that it is well-known and the possibility that more than one host in the victim's network would be involved in the attack. The "FTP Host" provides an anonymous FTP service that allows uploads and the "Target Host" provides a remote shell service that trusts the users on the "FTP Host."

1. In preparation, the attacker creates a file containing a valid remote shell (`rsh`) message such as
   `\0root\0root\0xterm -display bad.hacker.org:0.0`
   which means "I am the user `root` on the local computer, I wish to execute a command on the remote computer as the user `root`, and the command I wish to execute will open a terminal window from the remote computer on my screen."

2. The attacker scans for valid hosts in the target's network. For the purposes of our spatially distributed attack, assume the attacker discovers at least two host systems in the target's networks. (Reconnaissance)

3. The attacker scans for listening TCP ports on the target network's valid hosts. Assume the attacker discovers a vulnerable anonymous FTP server listening at TCP port 21 on the "FTP Host", and a remote shell daemon (`rshd`) listening at port 514 on the "Target Host." (Vulnerability Identification)

4. The attacker uploads the previously created file to the anonymous FTP server on the "FTP Host".

5. The attacker uses a "feature" of the FTP protocol to tell the FTP server to send the next download to port 514 on the "Target Host". Then the attacker issues a command to the FTP server that initiates a "download" of the file containing the rsh message. If the "Target Host" trusts the users on the "FTP Host", the remote shell daemon on the "Target Host" accepts the message and executes it due to an authentication vulnerability in the remote shell protocol. (Penetration & Control)

6. The "Target Host" opens a terminal window on the attacker's X Window server that provides the attacker with root privileged shell. The attacker may proceed with any number of activities, including: changing passwords or adding users; reading or changing any file on the system; erase traces of his/her presence; and install tools to sniff passwords, provide back doors for future access, and disguise his/her activities. (Embedding, Data Extraction & Modification, and Attack Relay)

3 Software Fault Trees

In this section, we briefly describe Software Fault Trees, discuss their use in specification of IDS and introduce the augmented SFTs for modeling intrusions.

Fault-trees have been used for security assessment, although not explicitly for IDS. Cited advantages include their "organization and preservation of informal discussions about security ramifications of design alternatives" (in argument trees [32]) and the possibility for efficient reuse of subtrees (in attack trees [54]). However, fault trees suffer from several limitations with regard to modeling "multiple attacker attempts, time dependencies, or access controls" as well as for not modeling cycles [47]. Hence, we augment the fault tree formalism to overcome some of these limitations. The resulting augmented fault trees provide a useful framework for modeling intrusions.

Two interesting aspects of the requirements phase of this prototype are as follows. First, the intrusion SFT models have been interpreted as specifications of the combinations of events that must be detected. That is, the IDS requirements are that each of the intrusion sequences possible in the SFT should be detected as soon (as low in the tree) as possible. The leaf events describe what components of a distributed system must be monitored by the software mobile agent. The interpretation of the SFT serves as the requirements specification.

Second, the intrusion SFTs have had to be extended with additional information specific to a particular system prior to their mapping into CPNs. This information is of three types:

- *Trust* indicates which members of a distributed system are trusted by other members,
- *Context* shows which events must all involve the same host(s) or connection(s), process(es) or session(s),
- *Temporal orderings* that give which events must be adjacent with no intervening events, or follow within a specific interval of time.

Without this additional system-specific information, the IDS yields many false positives, detecting intrusions where, in a specific network, there is none. That is, the set of events marked as intrusions by the SFT is a superset of the set of events that are actually intrusions in any specific network and must be constrained by additional network-specific knowledge. These topics are discussed in more detail below.

A *fault tree* is defined formally as a tree consisting of: 1) a hazard as the tree's root node, 2) basic events that contribute to the hazard as the tree's leaves, and 3) either AND gates or OR gates (representing Boolean AND or OR operations, respectively) as each of the intermediate nodes. The intermediate nodes determine the combination of basic events necessary for the root hazard to occur.

3.1 Software Fault Tree Analysis

We adapt standard Software Fault Tree Analysis (SFTA) technique. The root node in a fault tree represents a hazard (here, the intrusion) being analyzed. The necessary preconditions for the hazard are specified in the next level of the tree and joined to the root with a logical AND or a logical OR. Each precondition is similarly expanded until all leaves are primitive events. SFTA investigates the ways in which the hazard (root node) might occur. If a credible scenario (i.e., path through the tree or, more precisely, a cutset of the tree) exists, the SFTA identifies the nodes (i.e., which events) that should be monitored in order to detect intrusions.

SFTA [37] is used first to model intrusions and develop requirements for the IDS. SFTA is a natural fit as the IDS design resembles a tree where data is obtained at the leaf nodes, travels up through the internal nodes as data is correlated with other information, and rises to the root node when an intrusion is identified.

3.2 SFTA in the IDS Development Lifecycle

The augmented SFT specification are mapped into Colored Petri Nets (CPNs) [29] that serve as the design for the IDS. CPNs are a well-documented and frequently-used abstraction for modeling complex distributed

systems. They appear particularly suited for describing the gathering, classification, and correlation activities of an intrusion detection system.

The advantages of using SFT to model the specification, rather than using only CPNs, are fourfold:

1. *Usability*. The system support personnel who will be using the system typically have a great deal of knowledge about intrusions that must be elicited and represented systematically in order for the requirements for the IDS to be determined. Usually they are not experienced in, or interested in, formal modeling techniques such as CPNs. SFTs, on the other hand, are perceived as familiar, easy to use, and easy to teach and learn. For an IDS to be effective, the specification must be readily updatable. The usability of SFT is an advantage in eliciting and capturing knowledge about the requirements.

2. *Support for gradual refinement for defining intrusions*. SFT supports gradual development of intrusion specifications with different subtrees being developed to varying levels of detail, depending on the level of concern and the level of knowledge regarding that subtree. CPNs, on the other hand, are better at modeling a system at a uniform level of detail.

3. *Modeling the attack*. The augmented SFT defines the intrusion specification. It is from this representation that the requirements for intrusion detection are derived. The CPN models not the intrusion itself but the intrusion detection system, i.e., the design of the IDS.

4. *Countermeasures analysis*. The augmented SFT intrusion specification allows determination of countermeasures needed for an IDS to thwart attacks [22].

3.3 Augmented SFT

We define an *augmented Software Fault Tree* to be an SFT where leaf nodes may specify trust, ordering, and contextual constraints in addition to the basic events of a SFT. Specifically, constraint nodes are added to SFT to capture trust, order, and contextual relationships needed to develop satisfactory specification of intrusions.

The effect of adding constraint nodes may be demonstrated by considering the set E of all combinations of events that make the root node of a plain (unenhanced) SFT "true". The set $I \subseteq E$ of combinations of events that are *actual* intrusions must also make the root node of the augmented SFT "true". ($|I|$ ought to be much smaller than $|E|$.) The constraint nodes added to an augmented SFT should exclude the vast majority of the combinations of "false positive" events $E - I$. Thus the augmented SFT, enhanced with the constraint nodes described here, will more closely model the requirements for detecting the intrusion being modeled.

1. **Trust:** Members of a distributed system trust other members of the system. An example of trust currently used in our SFT is authorization. The trust constraint will have to be enhanced when additional intrusions are modeled that depend on other notions of trust.

   As an example of trust, a Network File System (NFS) server using AUTH_UNIX authentication usually trusts the source IP address and user ID in client requests. This allows a user on a trusted client host to access files on the file server without having to login to the server.

   Explicitly stating a trust relationship that is required for an intrusion to succeed provides information to an intrusion detection system developer that will help derive an accurate matching model for the intrusion. The syntax of this predicate is $Trusts((destination), (source))$ where *destination* is an ordered list of constants and variables describing the trusting destination, such as name of destination host, network, or netgroup and application and *source* is an ordered list of constants and variables describing the trusted source, such as name of source host, network, or netgroup and application.

   The $Trusts$ predicate is true if the *destination* assigns some trust to the *source*. Specifying trust relationships in this way allows matching relationships to be unified [53] with other trust relationships. A trust relationship is *true* if one of system's trust relationships successfully unifies with the relationship specified by the *Trusts* predicate. An example of such a trust relationship may be $Trusts((Rshd, targetHost), (sourceHost))$ which states that the remote shell daemon (*Rshd*) on a

targetHost trusts a *sourceHost*. By convention, elements beginning with upper-case letters are constants, and elements beginning with lower-case letters are variables.

2. **Context:** Certain combinations of intrusive events must occur in some common context. For example, a series of FTP commands and responses need to be grouped by a common network connection to an FTP server.

   In the following definitions of forms of context, each of the parameters (host, connection, user, or process) may be specified as a constant value or a variable. Network-related events may be related by events involving a single host, a pair of hosts, or a single virtual network connection.

   A single host that must be a common source or target for network events may be specified as a common context for intrusive events. The syntax for this constraint is $Context((:host\ Hostname)\ (FTNodeList))$ where *Hostname* is the name or address of a host or group of hosts and *FTNodeList* is the list of one or more SFT nodes to be included in the context. The predicate is true when the host identified by *Hostname* is involved in each node specified by the *FTNodeList*.

   Similarly, a pair of hosts that must be the source and target for network events may be specified as a common context for intrusive events using the syntax $Context((:hosts\ Hostname1\ Hostname2)$ $(FTNodeList))$ where *Hostname1, Hostname2* are names or addresses of hosts or groups of hosts. The predicate is true when hosts identified by *Hostname1, Hostname2* are involved in each node specified by the *FTNodeList*.

   Finally, a pair of hosts communicating using a virtual network connection that must be the source and target for network events may be specified as a common context for intrusive events using the syntax $Context((:conn\ Hostname1\ P1\ Hostname2\ P2)\ (FTNodeList))$ where *P1* and *P2* are names or numbers of network ports. The Context predicate is true when a network connection involving the endpoints identified by *P1* on *Hostname1* and *P2* on *Hostname2* are involved in each node specified by the *FTNodeList*.

   An authenticated user session on a host, such as via telnet, ssh, or ftp, may be a context for related events using syntax $Context((:user\ U\ App\ LH\ RH\ Term\ LT)\ (FTNodeList))$ where *U* is the name of a single user or group of users, *App* is the name of the method of access (e.g., telnet, ftp, etc.), *LH* is the name of the host to which the user is connected, *RH* is the name of a remote host or group of hosts, *Term* is the name of a terminal used for access (e.g., tty01), *LT* is the time of login, and *FTNodeList* is a list of one or more SFT nodes to be included in the context.

   Events corresponding to a process (an instance of a program in execution) may be a context for related events using the syntax $Context((:process\ PID\ Pg\ U\ Host\ ST)\ (FTNodeList))$ where *PID* is the identification number of the process, Pg is the program being executed, U is the set of user permissions, *Host* is the host on which the process executed, and *ST* is the time the process began executing. The context involves each node specified by the *FTNodeList*.

3. **Temporal Ordering and Intervals:** Events and conditions involved in an intrusion often must occur in a particular order. Explicitly specifying the event ordering excludes other non-intrusive permutations of events from being considered as intrusive. We use Allen and Ferguson's interval temporal logic [3] to develop temporal predicates.

   (a) **Occurs After**
   An event which takes place must make its node in the SFT true as long as the existence of that event may be combined with other events to make a parent node true. It seems an event's period may last as long as the context exists in which it may be evaluated. In this sense, "occurs after" is concerned only with the relative start of event's periods.

   "Occurs after" is the condition where one event's period is required to start after another event's period has started. The $Starts(i, j)$ primitive is true when periods i and j begin simultaneously. The $Meets(i, j)$ primitive is true when period i ends adjacent to the time where period j begins.

Let $Period(x)$ be the period that node x is true. Let $OccursAfter: Node, Node \rightarrow Boolean$ where $OccursAfter(i,j) \equiv \exists m\, Starts(Period(i), m) \wedge Meets(m, Period(j))$ meaning that the event or boolean expression indicated by the node *i* becomes true in the time prior to the time that the event or boolean expression indicated by the node *j* becomes true.

(b) **Adjacent Events**
Certain situations exist where an event must occur after another event within the same context with no intervening events. Let $ImmediatelyAfter: Node, Node \rightarrow Boolean$ where
$ImmediatelyAfter(i,j) \equiv OccursAfter(i,j) \wedge \neg(\exists n\, OccursAfter(i,n) \wedge OccursAfter(n,j))$
meaning the event or boolean expression indicated by the node *i* becomes true in the time prior to the time that the event or boolean expression indicated by the node *j* becomes true. No intervening events become true between *i* and *j*.

(c) **Interval**
An event may be required to follow another event within some amount of time. Let $StartOf(i)$ be the start of discrete time period i. The $Overlaps(i,j)$ primitive is true when period i overlaps period j. Then $InInterval: Node, Node, \mathbb{R} \rightarrow Boolean$ where
$InInterval(i,j,t) \equiv OccursAfter(i,j) \wedge Overlaps(\langle StartOf(Period(i)), StartOf(Period(i)) + t \rangle, j)$
meaning the event or boolean expression indicated by node *i* becomes true in time prior to the time that the event or boolean expression indicated by node *j* becomes true. Additionally, *j* must become true during the period specified by *t*.

4 Colored Petri Nets

In this section we introduce Colored Petri Nets and describe the transformation from augmented SFTs for intrusions to CPN templates for intrusion detection systems.

4.1 Colored Petri Nets Defined

CPNs are a powerful modeling technique for complex systems [29]. CPNs model state and action through the use of colored tokens (colors can be thought of as *data types*) which reside in *places* (or *states*). Tokens move from one place to another through transitions. Transitions allow tokens to pass if all input arcs are *enabled* (meaning tokens are available for each input arc). Tokens entering from multiple places may be merged (or unified) at transitions. Tokens leaving transitions may be duplicated to multiple destination places. CPNs may be organized in hierarchical fashion to allow reuse and top-down or bottom-up development.

In a graphical representation of a CPN, places are denoted by ovals or circles, transitions are denoted by squares or rectangles, and lines with arrows denote arcs. If a predicate or tuple is written next to an arc, a token must satisfy the predicate or unify with the tuple before it may pass through the arc. Token colors are defined at the entry point of each CPN in terms of tuples of standard values, such as strings or integers (tokens may also be defined as data structures). Places may be labeled with a particular color by an italicized label.

Formally, a Colored Petri Net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ satisfying the requirements: Σ is a finite set of non-empty types, called color sets, P is a finite set of places, T is a finite set of transitions. A is a finite set of arcs such that $P \cap T = P \cap A = T \cap A = \emptyset$, N is a node function $A \rightarrow P \times T \cup T \times P$, C is a color function $P \rightarrow \Sigma$, G is a guard function defined from T into expressions such that $\forall t \in T : [Type(G(t)) = Boolean \wedge Type(Var(G(t))) \subseteq \Sigma]$, E is an arc expression function defined from A into expressions such that $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$
where $p(a)$ is the place of $N(a)$[1]. I is an initialization function defined from P into closed expressions such that $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$." [29].

A hierarchical CPN consists of a set of CPNs arranged in a hierarchical structure. The two building blocks of hierarchical CPNs are substitution transitions and fusion places. Substitution transitions and fusion places

[1]The subscript "MS" indicates a multi-set, which Jensen defines as allowing "multiple appearances of the same element." [29, p. 66]

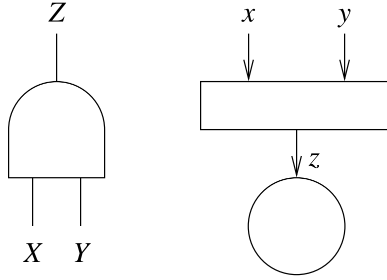

Figure 1: Unconstrained AND node with corresponding CPN

allow the construction of a hierarchical CPN by combining a number of non-hierarchical CPNs. A hierarchical CPN may be translated into a behaviorally equivalent non-hierarchical CPN, and vice versa. Hierarchical CPNs are important to our design of the IDS as they allow construction of detectors for components of attacks that can be composed into detectors for the complete intrusion.

CPNs have been applied to a variety of problems in security, networks, concurrent systems, VLSI chip design, and chemical manufacturing systems [30]. Petri Nets have also been applied to the safety domain [38], which is closely related to the security domain [52], and to IDS systems [34, 35].

Our work with modeling intrusion detectors as CPNs has shown that CPNs provide a formal foundation for the agent-based distributed IDS and allow analysis of the IDS for discovering inconsistencies between components of the system, finding ideal places in the monitored system for security improvements, and proving that certain attacks can not be successful if a system is changed so as to eliminate the identified vulnerabilities.

4.2 From Augmented SFT to CPN Templates

Colored Petri Net template intrusion detectors may be generated from augmented SFTs for intrusions to ensure correctness and correspondence between a requirement specification based on SFT and a design using CPNs. The constraints added to an augmented SFT to describe the ordering relationships between nodes requires special handling to develop accurate CPN templates from augmented SFT.

Leaf nodes in the augmented SFT for intrusions correspond to basic events in the system which must be detected. Leaf nodes then correspond to token source places in the CPN. The token source places produce a new token each time the basic event takes place. Tokens generated by token source places must have sufficient descriptive information so that tokens may be matched and unified to satisfy any trust, context, and ordering constraints that exist in the augmented SFT.

AND nodes in the SFT are of special interest in intrusion models. Semantically, when all child nodes of an AND node in a SFT are true, the AND node is true.

AND Nodes without Ordering Constraints An AND node unconstrained by an ordering in an SFT corresponds to a transition and outgoing place pair in a CPN. An AND node with n inputs translates to a transition with n incoming arcs. Each incoming arc comes from either a token source place of an SFT leaf node, or the outgoing place of an SFT gate node. Figure 1 illustrates the correspondence between an AND node and its equivalent CPN transition/place pair, where:

$$X = \begin{cases} 1, & \text{if } x \in D_x \\ 0, & \text{otherwise} \end{cases}$$

$$Y = \begin{cases} 1, & \text{if } y \in D_y \\ 0, & \text{otherwise} \end{cases}$$

$$Z = \begin{cases} 1, & \text{if } (x \in D_x) \wedge (y \in D_y) \\ 0, & \text{otherwise} \end{cases}$$

X and Y are the binary inputs to the AND gate, and Z is the binary output of the AND gate. x and y are the incoming tokens to the CPN transition, where D_x and D_y are the domains of x and y, respectively. z is the output token from the CPN transition.

Tokens leaving the transition must be unified such that they satisfy any related trust and context constraints that exist higher in the augmented SFT. An examination of the trust and context constraints that are connected to branches along the path to the root in the minimum cut of the augmented SFT will identify the constraints for the events described by the incoming tokens. The designer must construct the unifying expressions so that the related elements in the output token(s) satisfy the constraints. For example, if a constraint exists in the minimum cut that requires two augmented SFT nodes to be related by a common TCP network connection context, the tokens must be unified using the elements of the TCP quad (source host, source port, destination host, and destination port) that uniquely identifies a TCP connection; this would satisfy the connection context constraint node in the augmented SFT.

AND Nodes with Ordering Constraints Nodes connected to an AND node in an augmented SFT may have an attached constraint that requires the nodes to become true in some particular order. Two cases exist: first, nodes may be required to become true in order, but intervening events may occur; second, nodes may be required to become true in order with no intervening events.

To support ordering, CPN tokens are required to contain times or sequence numbers. If event a occurs before event b, the timestamp in the token representing event a must be less than the timestamp in the token representing event b, and no two events may have identical timestamps. Likewise, if event a occurs before event b, the sequence number in the token representing event a must be less than the sequence number in the token representing event b, and no two events may have identical sequence numbers.

Literal wall-clock times used for comparisons are a problem when the times are obtained from different computers in a distributed system [60]. Each computer has its own notion of the current time, and computer clocks tend to skew at different rates. We assume that the clocks are kept synchronized and the skew δ between a computer's clock and the actual time is very small. As an implementation detail, the intrusion detection system may itself synchronize the clocks and monitor the measured difference δ_m between clocks. The IDS may have an established maximum skew δ_{MAX} and may consider any $\delta_m > \delta_{MAX}$ to be an intrusion. In addition, the implementation may include δ in its comparisons between timestamps. The comparison $t_1 + \delta < t_2 - \delta$ yields a tight bound on two events, which may result in false negatives. The comparison $t_1 - \delta < t_2 + \delta$ yields a loose bound on two events, which may result in false positives.

Sequence numbers are maintained per context and no comparison may be made between sequence numbers across contexts.

The addition of temporal ordering to augmented SFT and the associated representation of time information in event tokens enables temporal reasoning.

1. *Occurs After:* The case "occurs after" covers the situation where augmented SFT nodes must become true in a particular order.

   Figure 2 shows an example of an AND node constrained such that node y must occur (become true) after node x becomes true, where:

$$X = \begin{cases} 1, & \text{if } x \in D_x \\ 0, & \text{otherwise} \end{cases}$$

$$Y = \begin{cases} 1, & \text{if } y \in D_y \\ 0, & \text{otherwise} \end{cases}$$

$$Z = \begin{cases} 1, & \text{if } (x \in D_x) \wedge (y \in D_y) \wedge (time1 < time2) \\ 0, & \text{otherwise} \end{cases}$$

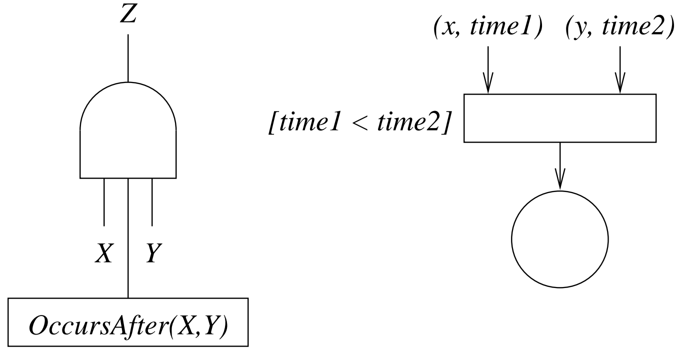

Figure 2: AND node, constrained by "Y after X", with corresponding CPN

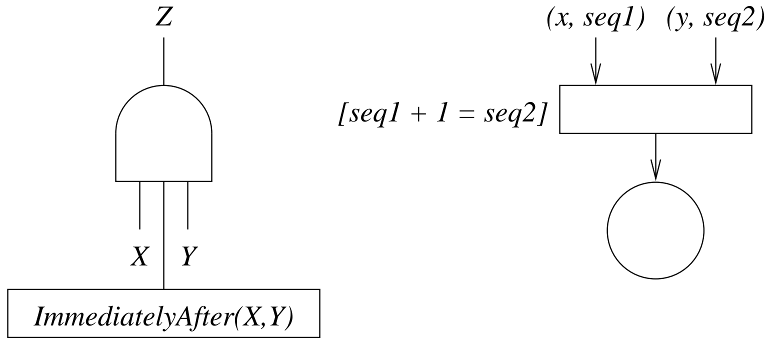

Figure 3: AND node, constrained by "Y immediately after X", with corresponding CPN

In Figure 2, $time1$ and $time2$ denote the timestamps for events x and y, respectively. The related CPN segment shows that a token for event x must have a smaller timestamp than the token for event y. The significant differences between Figures 1 and 2 are the addition of time information to the tokens, and the guard on the transition that enforces the ordering on the token's time.

Timestamps in the "occurs after" case may be either wall-clock time or sequence numbers.

2. *Immediately After:* The case "occurs immediately after" covers the situation where augmented SFT nodes must become true in a particular order. Intervening events may not occur.

   Figure 3 shows an example of an AND node constrained such that node y must occur (become true) immediately after node x becomes true, where:

$$X = \begin{cases} 1, & \text{if } x \in D_x \\ 0, & \text{otherwise} \end{cases}$$

$$Y = \begin{cases} 1, & \text{if } y \in D_y \\ 0, & \text{otherwise} \end{cases}$$

$$Z = \begin{cases} 1, & \text{if } (x \in D_x) \wedge (y \in D_y) \wedge (seq1 + 1 = seq2) \\ 0, & \text{otherwise} \end{cases}$$

   In Figure 3, $seq1$ and $seq2$ denote the sequences numbers for events x and y, respectively. The related CPN segment shows that a token for event y must have the timestamp immediately following the

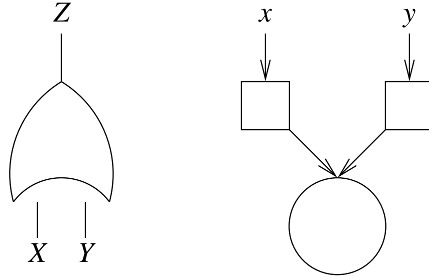

Figure 4: OR node with corresponding CPN

timestamp for event x, implying that discrete timestamps (sequence numbers) are necessary for the operation of this CPN segment.

OR Nodes When any of the child nodes of an OR node in an augmented SFT is true, the OR node is true.

An OR node in an augmented SFT corresponds to a set of transitions and outgoing place pair in a CPN. An OR node with n inputs translates to n transitions, each having 1 incoming arc. Each incoming arc originates in either a token source place based on an augmented SFT leaf node, or the outgoing place based on an augmented SFT gate node. Figure 4 illustrates the correspondence between an OR node and its equivalent CPN transitions and place, where

$$X = \begin{cases} 1, & \text{if } x \in D_x \\ 0, & \text{otherwise} \end{cases}$$

$$Y = \begin{cases} 1, & \text{if } y \in D_y \\ 0, & \text{otherwise} \end{cases}$$

$$Z = \begin{cases} 1, & \text{if } (x \in D_x) \vee (y \in D_y) \\ 0, & \text{otherwise} \end{cases}$$

As with constraints on events in an AND node, tokens leaving the transition for an OR node must be unified such that they satisfy any trust and context constraints that exist higher in the augmented SFT.

4.3 Generation of Token Definition

Defining token types in converting an augmented SFT into CPN is more difficult than generation of places and transition since there is a fundamental difference between augmented SFT and CPN. Augmented SFTs describe what constitute a hazard at a conceptual level and give less details to the events than in data level. But for CPNs, especially for CPNs that are used to generate a program, more detailed description of what constitutes an event is needed.

For a leaf node in an augmented SFT (which corresponds to a token source place) we need to add some explanation to the event. When the augmented SFT is translated to CPN, we then have the necessary information about what constitutes the event and what kind of token should be fired by the corresponding event. For example, if the event is FTP_PORT_OK, we may add an explanation in the representation of the augmented SFT like $type = "RESPONSE", src_port = "21", value = "2xx"$. Then when the token source place is generated, we can specify enough information to describe the event in the token and enable further token matching and unification.

4.4 Automatic Translation from Augmented SFT to CPN Templates

Based on the translation rules given above, an automatic translation procedure has been designed and implemented. The procedure makes use of XSL and XML definitions of augmented SFTs and CPN templates. Document Type Definitions (DTDs) have been developed for augmented SFTs and CPN templates.

The process has the following steps:

1. Convert an augmented SFT to its XML equivalent using the translation rules described above.
2. Translate the augmented SFT XML to the XML of the corresponding CPN template using an XSL transformer program, such as Visual XML Writer.
3. View and validate the CPN corresponding to the resulting CPN template XML from step 2 using a CPN tool, such as Design/CPN [11]. During this step, the CPN can be optimized to improve efficiency.

Figure 6 presents a CPN generated automatically from the augmented SFT in Figure 5 for the FTP Bounce Attack using the XSL technique.

5 From CPNs to an Agent-Based Implementation

Similar to the algorithmic approach for creating CPN templates from augmented SFT, an algorithmic approach to creating an agent-based implementation from the set of CPN templates has been developed.

By using the translation algorithm to convert CPN designs to code, we can be certain that the code implements the CPN design. Also, if the translation algorithm preserves CPN semantics, any analysis performed on the CPN design also applies to the implementation. Finally, creating an implementation in code allows the developer to improve performance over the execution of a general CPN.

A distributed implementation of the CPN model using software mobile agents can provide a reliable, robust, and efficient IDS [27]. Our implementation provides useful information to the security analyst in the form of a trail of transitions through which CPNs passed. [57] provides further details regarding the MAIDS implementation.

The MAIDS prototype uses the Voyager agent platform, version 3.2, from Objectspace [46] [2]. An agent is an instance of a Java class, which may be created either locally or remotely. Additionally, Voyager supports code mobility of two types. One agent can move another agent between hosts, or an agent can request its own migration. The use of Voyager is not central to the MAIDS design. Other agent platforms that have been considered for use include Grasshopper from IKV++ [24] and SMART [62].

5.1 Java-based Conversion Algorithm

Each CPN place maps to an agent in our implementation. The algorithm for translating a distributed CPN to an implementation of software mobile agents in Java is as follows:

1. For each leaf place node in a CPN, instantiate an agent that extends the `DataPlace` class;
2. For each internal place node, instantiate the `Place` class with a unique label;
3. For each leaf transition node, instantiate an agent that extends the `MobileTransition` class;
4. For each internal transition node, instantiate an agent that extends the `StationaryTransition` class;
5. Main console instantiates an `AlertPlace` agent for the root node. Refer to it with label *alert* in all transitions that have an outgoing edge to it.

[2] ObjectSpace spun off a new company called Recursion Software Inc. in 2001 to handle Voyager, http://www.recursionsw.com

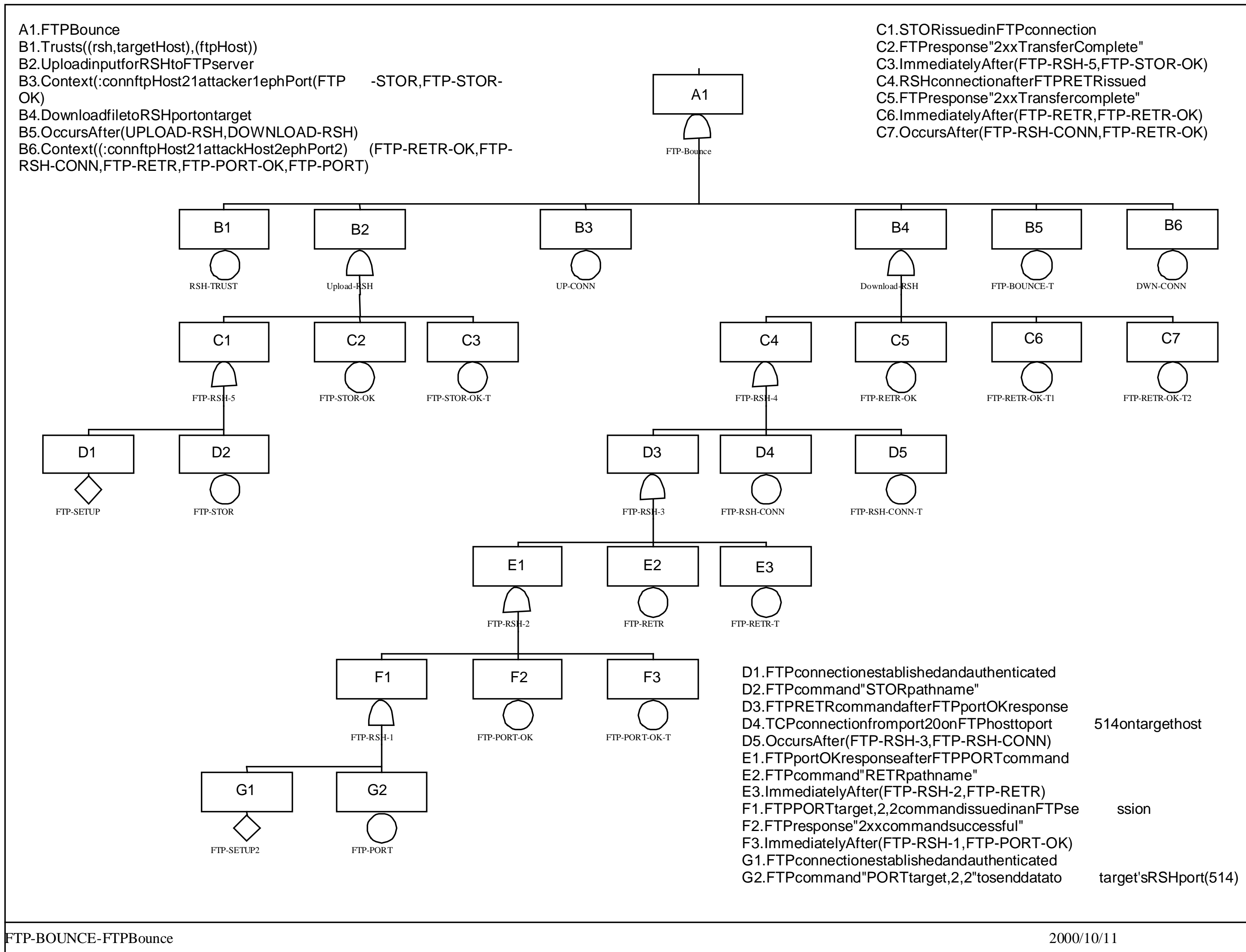

Figure 5: Fault tree for FTP bounce attack, with constraints

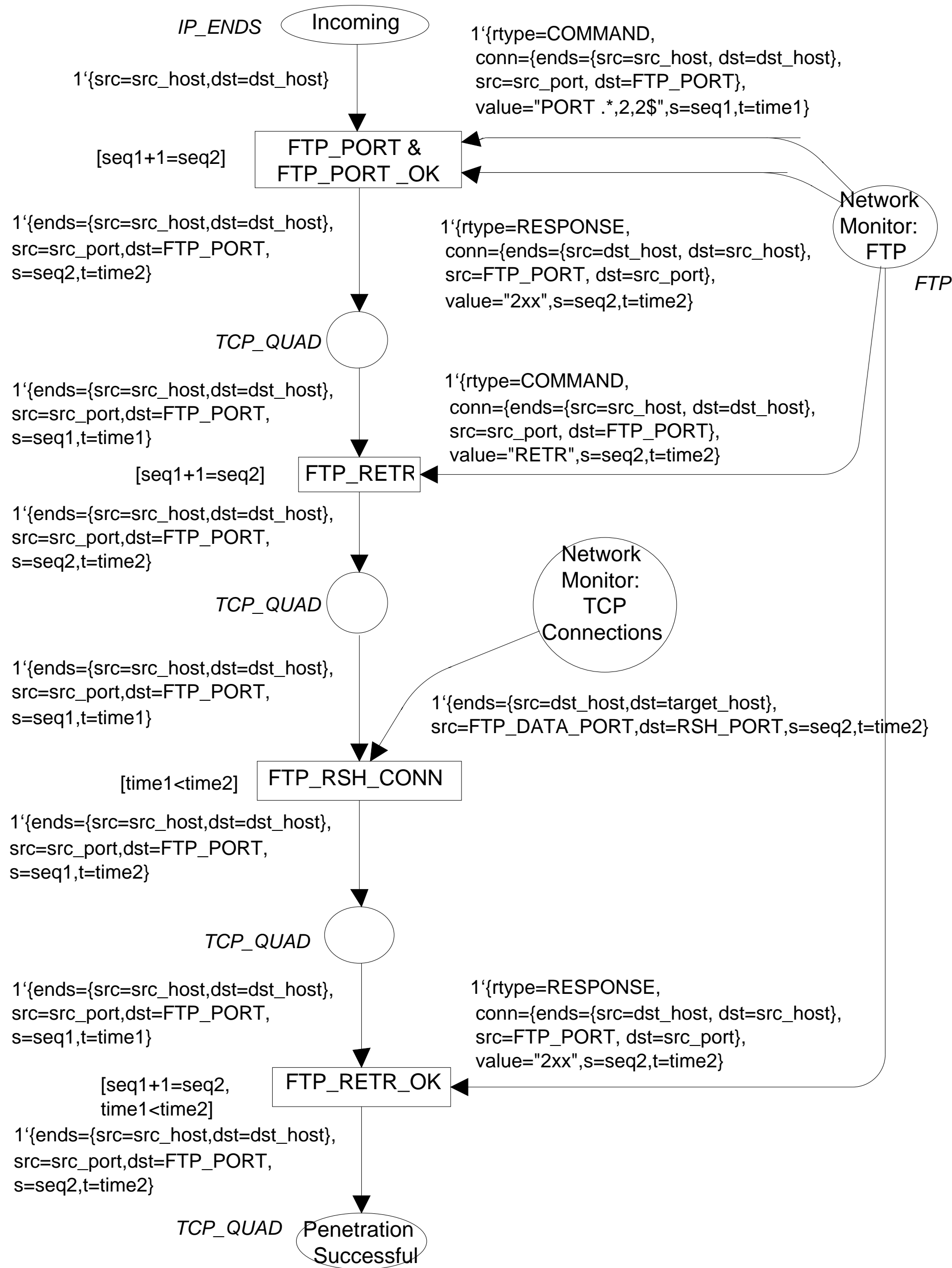

Figure 6: FTP Bounce Attack Penetration CPN generated automatically from augmented SFT via XSL

The CPN arcs, which constitute the structure of the agent network, are not maintained centrally. Each transition knows about the places to which it is immediately connected, and places know nothing about the agent network structure.

The algorithm preserves the CPN semantics in the implementation and allows for efficient execution. Performance can be increased over the execution of a general CPN by optimizing code segments to fit specific intrusion detection applications. For example, matching and unifying tokens is computationally intensive, at least $O(n^2)$ in the general case; but in cases where the number and types of tokens are known in advance, faster algorithms can be used. If a place holds tokens of a single type, an implementation could match tokens in $O(n \log_2 n)$ time based on a binary search or $O(n)$ time based on hash tables. Such enhancements could be made to the *Transition* superclass so that they would not be a burden to the end user, but they are not part of the current MAIDS prototype.

Every agent must provide, strictly for debugging purposes, an `agentName()` method returning an identifying string. Beyond this, agents have specific requirements as follows.

1. *Required methods for data place agents:* The responsibility of a data place is to generate fresh tokens from whatever information is locally available. It must implement a *work()* method, taking no arguments and returning a *TokenBag*. This method will be called periodically from the *DataPlace* superclass code.

2. *Required methods for transition agents:* Transitions are much more complex, embodying as they do all of the logic of the agent network. But most of the complexity is hidden in superclass code. Three additional methods must be implemented: $String[]\ sources()$, $String[]\ tokenSpec()$, $Token[]\ unify\ (Token[]\ sourceTokens)$.

Of these, only the last is nontrivial to write. `sources()` should return an array of the labels of the places that have arcs to this source; `tokenSpec()` should return an array of token colors, and determines what kind of input the `unify()` method will see. For instance if `tokenSpec()` returns the array {`"blue"`, `"red"`}, then whenever `unify()` is called it will be given an array containing one blue token and one red token, in that order. `unify()` is then responsible for deciding whether those tokens should be unified. If so, it returns a new array of tokens; otherwise it returns null.

Notably, there is no explicit `destinations()` method. This information is placed in the tokens themselves via an argument to the `Token` constructor. In the case where the token is created by a data source place, there is no delivery, so the source place itself is given as a destination; in the case where a transition is creating a new token, the destination is determined by an outgoing arc on the CPN.

Behind the scenes, the `Transition` superclass is responsible for iterating through the tokens available from the places given in `sources()`, retrieving from there sets of tokens satisfying the description given in tokenSpec, and presenting them to the agent class as an argument to `unify()`. If it gets an array of tokens in return, it delivers them to their designated places and deletes the source tokens; otherwise it leaves the tokens where they were found.

5.2 Preservation of CPN Semantics

The MAIDS representation of a CPN as a network of Java objects satisfies the CPN properties listed in Section 4.1 as follows:

1. Each type is a color in set Σ.

2. Each instance of a `Place` is an element in set P.

3. Each instance of a `Transition` is an element in set T.

4. Arcs A between places and transitions are encoded in the transitions (when proceeding from place to transition) and tokens (when proceeding from transition to place); they are finite in number and satisfy the requirement as they are distinct from places and transitions.

5. The encoding of arcs as described above defines the node function N.

6. The assignment of color in the `Token` class constructor defines the color function C.

7. A `unify()` method implements the guard expressions in G for each transition in T.

8. The `unify()` method also implements the arc expression E for each arc $a \in A$.

9. A trivial initialization function I makes each place begin with no tokens.

The implementation of transitions and places may impose additional constraints not present in the CPNs so as to obtain efficiencies for particular expected token colors; as long as only expected token colors exist in the places, the CPN semantics are satisfied by the implementation.

5.3 Algorithm For Translating CPN Design Templates to Agent Implementation

The MAIDS uses a distributed agent-based system to detect intrusions. If the CPN model of intrusion detection is expanded to include multiple data source nodes (which are simply duplicated places that provide the same token colors to transitions), and transitions are given mobility, the result is a distributed CPN (DCPN). In our design, transitions are selected for mobility based on their need to visit different sites in the distributed system to collect tokens from duplicated place nodes for matching. The places visited by the transitions are defined dynamically through the user interface, corresponding to the nodes that are being monitored by MAIDS.

The previous section examined the implementation of a CPN as Java code. This section further details the implementation of a CPN as agents in a distributed system.

1. **Node Categories:** The IDS CPN design resembles a tree where data is obtained at the leaf nodes, feeds up through the internal nodes, and finally reaches the root node when an intrusion is identified. Tokens in the IDS CPN represent information that, as tokens "rise" through the tree, is correlated with other information to identify intrusions.

   Source places (places which have no incoming arcs) are considered *leaf places*. The transitions adjacent defined to leaf places are considered *leaf transitions*.

   Sink places (places which have no outgoing arcs) are considered *root places*. The Alert place is currently the single root place in the CPN IDS design.

   Internal places and *internal transitions* are the remaining places and transitions, respectively, in the CPN IDS design.

2. **Leaf Places and Transitions:** Raw audit data of various types and formats is obtained from monitored systems for the IDS. Data cleaning agents have been developed to read and process the raw audit data for use by the IDS. The data cleaning agents correspond to the leaf places and transitions in the CPN design.

   Leaf places and transitions are duplicated at each monitored system to manage the constant process of data retrieval and cleaning.

   The leaf places (data cleaners) are agents that remain in a single location to obtain raw data, such as that available from log files. In the current MAIDS implementation, the leaf places are instantiated separately on each host by the operator, where they will remain stationary for the duration of their activity. Next generation of MAIDS would allow the console to dispatch the leaf nodes to the monitored host and allow the console to recall the agent to replace it with an updated agent or cease monitoring. The leaf places perform minimal processing and do not place a substantial resource load on the monitored systems.

Leaf places are an instance of places in the MAIDS DCPN implementation that require customized coding to perform operating-system specific data gathering and cleaning tasks. Nearly all other places are generic, passive containers of tokens.

Leaf transitions (data gatherers) are software mobile agents that travel between monitored systems to obtain tokens. Currently, single instances of each leaf transition perform the data gathering duties, but in the future, multiple instances of each leaf transition could cooperate to gather data in a large distributed system.

Informally, the leaf transitions perform the first level of data gathering and filtering in the IDS. Formally, the leaf transitions perform the token matching and unification specified by the CPN IDS design.

3. **Internal Places and Transitions:** Internal places act as passive containers for tokens. Internal places are not duplicated; a single instance exists and accepts tokens from all (possibly mobile) transitions connected to it. Internal places currently reside at the machine running the console, but they could be given mobility if it becomes advantageous.

   Internal transitions are similar to leaf transitions in that they apply token matching and unification rules to tokens as they are obtained from incoming places and sent to outgoing places. Like internal places, internal transitions are statically positioned at the machine running the console. Internal transitions could be given mobility if advantages are found.

4. **Root Place:** The root of the CPN IDS design is the alert place. It acts as a passive container, but when a token is added to the alert place, the IDS console interprets the token and displays it. Transitions are required to set an urgency level parameter in tokens for use by the IDS console. Tokens are sorted on the IDS console display by their urgency and then by their arrival time.

5. **The IPlace Interface:** The `IPlace` interface specifies four methods: `void storeToken(Token t)`, `TokenBad getTokens()`, `boolean lock()` and `void unlock()`.

   All `Place` agents in the network, except the data source (leaf) places, are instances of final classes. As a result, the end implementer is never responsible for any of these methods. They are called by transition agents, but in superclass (`Transition`) code so that they are invisible to the implementer. Additionally, `DataPlace` superclass code uses `storeToken()`. The `lock()` and `unlock()` methods allow a transition to atomically examine and either replace or remove tokens from several places.

5.4 Testing CPN design

A set of use cases (positive and negative examples of intrusions, i.e. set of paths in CPN leading to successful or unsuccessful intrusion) were developed to test the intrusion detection system requirements. The CPN design was tested using the use cases to observe the behavior of the CPN and verify correct functionality. Equivalence classes may be used to test representative samples from groups of intrusions to reduce the testing effort [49].

Since the requirements model is less detailed than the CPN and may not be as expressive as a CPN model, the CPN design further constrains the sets of events that will be identified as intrusions. Thus, some use cases that are identified by the requirements as intrusions will not be considered intrusions by the CPN model and the intrusion detection system implementation. Each use case must be annotated to describe whether the requirements and/or design will identify the use case as an intrusion.

CPNs design was tested with the following methods:

1. Interactive simulation - Execute a CPN model in a way similar to interactively debugging a program.

2. Automatic simulation - Investigate functional correctness and performance of a CPN model by executing a CPN at full speed.

3. Creating occurrence graphs - Determine reachability of nodes in a CPN model.

4. Place invariants - Prove user-specified predicates to be satisfied for all reachable system states to prove properties such as absence of deadlock.

Place invariants in particular may be useful for the intrusion detection CPN design, as they may allow invariants to be derived from requirements and verified in the CPN design. For example, a place in an FTP bounce attack detector of Figure 6 may have an "FTP RESPONSE" token only if there exists a matching "FTP COMMAND" token in the CPN, since a command must be issued to receive a response.

Interactive simulation has been performed by building CPNs and simulating their execution in the Design/CPN tool [11] using positive and negative examples of intrusions. Automatic simulation has been performed indirectly by building an implementation of CPNs in Java and executing it.

5.5 Detecting FTP Bounce Intrusion

FTP bounce detection was tested using a script to launch the attack from a host outside the local network. Because real intrusion data for this attack was not readily available in the form of network traces, we mixed normal and malicious sessions to simulate attack under significant network traffic conditions. An upload of a one-line text file followed by a download of the same file was our model of a "normal" session, and was chosen for its superficial similarity to an instance of an FTP bounce attack. The normal session scenario, like the attack scenario, was made repeatable using our scripts. A Perl script invoked these scripts to run 50 ftp sessions sequentially; the sessions numbered 2 and 49 were malicious, and the rest were normal.

Two monitored hosts were attacked, one as relay and the other as target. A third machine served as the host console. The relay host was running a modified version of the wu-ftp server. Changes were made to source code file `ftpcmd.y` to blindly enable *PORT* commands regardless of source or destination. While this very vulnerable server was active for testing, packet filtering was kept in place to discourage real attacks from outside our laboratory's domain. Also, the target host's RSH service was not made vulnerable; instead, RSH service was disabled and a proxy was set up to watch port 514 and echo its traffic to a terminal window. By these measures, all the essential events could appear as a real attack, but with minimal danger of our test systems being compromised.

When run in isolation, a scripted attack was detected typically between 2 and 5 seconds of its completion. This time disparity was to be expected because of the discrete actions of the agents and the randomized delays that were artificially inserted.

In tests of the 50-session ftp sequence, the two malicious sessions were reliably detected (*i.e.,* no false negatives) with no false positives.

Details of the alert tokens from one such test are shown in Figure 7. These text presentations appear at the analyst's console when a token is selected from the alert list panel and the "details" button is pressed. The hierarchical indentation scheme reflects the history of unifications that led to the creation of a token. (Note: the recorded creation times of the `RSH_PORT` and `FTP_BOUNCE_ATTACK` tokens reflect a clock skew between the monitored hosts, since token timestamps depend on the machine where unification actually takes place.)

Although both attacks were correctly identified out of the scripted ftp sessions, the later attack took significantly longer to detect. Studying the contents of the two tokens, it is apparent that the bottleneck is in the creation of the `FTP port & retr` token, which is the job of the complex transition that was created as a result of node reduction. After the first attack, the `FTP port & retr` token appears at 20:26:24, 4 seconds after all four contributing tokens are available. But in the second attack this disparity is larger: the contributing tokens are available by 20:29:14 and are not unified into a `FTP port & retr` token until 19 seconds later, at 20:29:33. The difference in performance is accounted for by the fact that each of the 48 intervening normal ftp sessions produced `FTP_PORT_OK`, `FTP_RETR`, and `FTP_RETR_OK` tokens, all of which had to be processed by the `FTPB_relay_MT` agent in every possible combination.

The test conditions are such that the `FTPB_relay_MT` transition unifies in $O(n^4)$ time, where n is the maximum number of tokens of any color. It would seem that a small n is required to prevent the

```
--- Token: [urgency 9] [type FTP BOUNCE ATTACK] ---
FTP BOUNCE ATTACK [0] Mon Apr 02 20:28:27 CDT 2001
   FTP port & retr [0] Mon Apr 02 20:26:24 CDT 2001
   > 986261179000 986261180000
      FTP_PORT [585] Mon Apr 02 20:26:19 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT requested
      FTP_PORT_OK [586] Mon Apr 02 20:26:19 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT succeeded
      FTP_RETR [587] Mon Apr 02 20:26:19 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR requested
      FTP_RETR_OK [588] Mon Apr 02 20:26:20 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR succeeded
   RSH_PORT [0] Mon Apr 02 20:28:21 CDT 2001
   > tcp://judge.cs.iastate.edu:8000/tcp monitor: rsh connection from ftp

--- Token: [urgency 9] [type FTP BOUNCE ATTACK] (1) ---
FTP BOUNCE ATTACK [1] Mon Apr 02 20:31:37 CDT 2001
   FTP port & retr [2] Mon Apr 02 20:29:33 CDT 2001
   > 986261354000 986261354000
      FTP_PORT [816] Mon Apr 02 20:29:14 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT requested
      FTP_PORT_OK [817] Mon Apr 02 20:29:14 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT succeeded
      FTP_RETR [818] Mon Apr 02 20:29:14 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR requested
      FTP_RETR_OK [819] Mon Apr 02 20:29:14 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR succeeded
   RSH_PORT [7] Mon Apr 02 20:31:15 CDT 2001
   > tcp://judge.cs.iastate.edu:8000/tcp monitor: rsh connection from ftp
```

Figure 7: Alert tokens from node-reduced test

`FTPB_relay_MT` transition from wasting a great deal of the local host's CPU time, even if the observed total detection times of under 30 seconds were otherwise deemed acceptable.

To correct this, the first impulse might be to test an agent system based on the original CPN before node reduction. But to facilitate component reuse it is desirable to keep the `FTP port & retr` token intact. Therefore the unreduced CPN is rearranged as in Figure 8 for a second test with the same 50-session ftp script.

As in the previous test, both attacks were detected and there were no false positives. The attack token details are shown in Figure 9.

The first attack is detected more slowly than before, but the second more quickly (compare creation times of the `FTP_RETR_OK` and `FTP port & retr` tokens, to witness that the bottleneck is relieved). Total detection times from these tests, calculated from reception of the attack egg to creation of final alert token, are summarized in Figure 10.

We see that node reduction, in addition to simplifying the layout of the agent system, decreases the constant agent communication and migration overhead, and so performance improves under light-traffic conditions. But when large numbers of tokens accumulate in a short period of time, complex transitions perform poorly and overall performance suffers.

This performance analysis is by no means exhaustive but gives a general indication of the effect of node reduction.

6 Other intrusion scenarios

FTP Bounce attack example demonstrates all development stages of intrusion detection system (from specification of intrusion to design and implementation of the intrusion detection agents) and the final result of this process - actual detection of an attack. In addition to this intrusion, we have also tested several other attack scenarios in wired and wireless settings. For brevity we only include the descriptions of those scenarios.

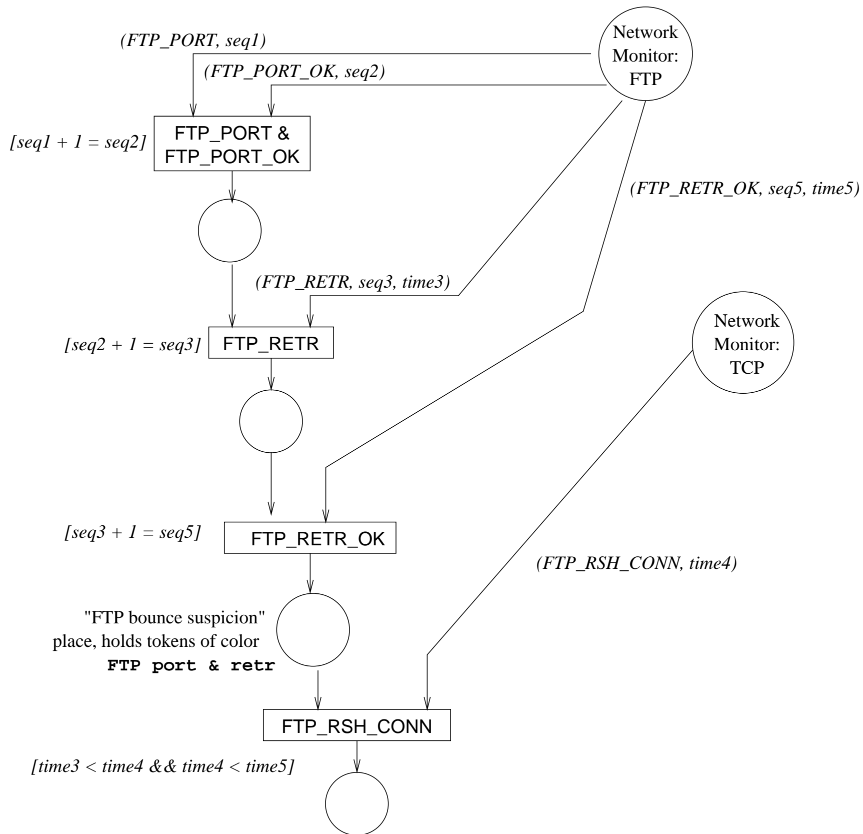

Figure 8: FTP bounce CPN, rearranged for later component reuse

```
--- Token: [urgency 9] [type FTP BOUNCE ATTACK]  (2) ---
FTP BOUNCE ATTACK [2] Mon Apr 02 22:31:43 CDT 2001
   FTP port & retr [1161] Mon Apr 02 22:35:44 CDT 2001
   > 986268932000  986268932000
      col 2 [1160] Mon Apr 02 22:35:37 CDT 2001
      > 986268932000
         col 1 [1159] Mon Apr 02 22:35:35 CDT 2001
            FTP_PORT [1158] Mon Apr 02 22:35:32 CDT 2001
            > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT requested
         FTP_PORT_OK [1159] Mon Apr 02 22:35:32 CDT 2001
         > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT succeeded
      FTP_RETR [1160] Mon Apr 02 22:35:32 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR requested
   FTP_RETR_OK [1161] Mon Apr 02 22:35:32 CDT 2001
   > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR succeeded
RSH_PORT [35] Mon Apr 02 22:31:30 CDT 2001
> tcp://judge.cs.iastate.edu:8000/tcp monitor: rsh connection from ftp

--- Token: [urgency 9] [type FTP BOUNCE ATTACK]  (3) ---
FTP BOUNCE ATTACK [3] Mon Apr 02 22:34:25 CDT 2001
   FTP port & retr [1394] Mon Apr 02 22:38:23 CDT 2001
   > 986269090000  986269091000
      col 2 [1393] Mon Apr 02 22:38:17 CDT 2001
      > 986269090000
         col 1 [1392] Mon Apr 02 22:38:13 CDT 2001
            FTP_PORT [1391] Mon Apr 02 22:38:10 CDT 2001
            > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT requested
            FTP_PORT_OK [1392] Mon Apr 02 22:38:10 CDT 2001
            > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp PORT succeeded
         FTP_RETR [1393] Mon Apr 02 22:38:10 CDT 2001
         > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR requested
      FTP_RETR_OK [1394] Mon Apr 02 22:38:11 CDT 2001
      > tcp://luke.cs.iastate.edu:8000/ftp monitor: ftp RETR succeeded
   RSH_PORT [40] Mon Apr 02 22:34:09 CDT 2001
   > tcp://judge.cs.iastate.edu:8000/tcp monitor: rsh connection from ftp
```

Figure 9: Alert tokens from test without node reduction

	low-traffic attack	high-traffic attack
With node reduction	6	22
Without node reduction	13	16

Figure 10: Detection times (in seconds) from Figures 7 and 9

6.1 Denial-of-service attack (DoS)

Denial-of-service attack is an off-line attack that was adapted from [1]. The goal of this scenario is to ensure MAIDS agents can gather and correlate data from multiple hosts to find intrusions. It uses pre-conditions and post-conditions to link events. As alerts are generated by an IDS, they are compared against rules in a database to determine if a correlation can be made. If one is made, a hyper-alert is generated to represent the alerts involved.

Test data for this scenario was taken from [1] and arbitrarily divided between multiple files in such a way that each file is similar in length, and no two of these files have data from the same line in the original file. The new files are then placed on different hosts in the network. An agent, created to visit each node in the network, collects alert data from these files, and attempts to link events using pre-conditions and post-conditions.

The correlation is done in a decentralized fashion. Using pre-condition and post-condition rules (stored in arrays in the agent code), 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. Once the agent has visited all hosts in the network, it writes all hyper-alerts to the console machine for analysis by an administrator.

6.2 Nmap scan

The second considered attack is a distributed Nmap scan[16]. 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. Nmap scan is considered as an attack as it is likely to be a first step in carrying out an intrusion. As such, hosts suspected of executing these scans are often disconnected from the network.

The attack is performed by sending SYN packets to targeted hosts. RST packet received from the target, instead of an ACK packet, is an indication that the port is not active and cannot be used in a later intrusion attempt. To 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.

In this scenario, the agent travels between three hosts. The attacker, on a third machine, performs a randomized port scan on ports 20 to 150 on the other two hosts. The job of the agent is to detect what appears to be completely random (and few in number) port activity on each host, and to correlate the aggregated results to decide if an Nmap port scan is being carried out on the network.

For this experiment, the agent correlated the events if similar alert patterns are found on all hosts with the same source address as seen earlier. If enough of these events are correlated, the agent takes a predefined action. In this scenario, once 100 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 to alert users of the scan.

6.3 Distributed real-time attack

The third attack demonstrates a distributed attack resulting in a compromised system. It was tested on the network of four hosts. Host A and T are the attacker and the target machine respectively. C1 and C2 are nodes which have been compromised by an attacker and are used to carry out the attack on T.

The first step of this attack is to perform a port scan from C1 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 and launches a more intrusive attack from C2 against it. The intrusive attack that was implemented is the Nachi worm [2]. This worm starts by sending an ICMP ping to the victim machine, and if reply is received it attempts to propagate.

In this scenario, agent is looking for three things: a port scan, a machine trying to obtain banner information from a service, and the virus signature. The port scan is detected by the same method as described in Scenario 6.2. When the agent detects a machine trying to obtain banner information from a local service on the current host, the agent checks the service port against previously scanned ports. If the port has been previously scanned, the agent correlates those two events.

As the agent moves from host to host, it also carries with it a signature for the Nachi worm. This signature is a pattern for detecting traffic from the worm. When the pattern is found in the log file, agent also checks previously scanned ports for the signature. If there is a match, the events are correlated. Once the worm has been detected, the agent raises its alert level status and prints the alert, along with the correlated events, to the screen of each host.

7 Related Models

An early intrusion detection approaches proposing detection of intrusions through anomalous user behavior were introduced by Anderson [6] and Denning [14]. Since then substantial amount of research attention has been directed into intrusion detection area [5, 7, 8, 12, 48, 31]. One of these intrusion detection techniques is misuse detection approach, although widely employed for detection of known attack patterns, also shown to have potential of recognizing unknown intrusions [40].

In the past two decades a number of misuse techniques have been proposed. Among these are methods based on rule-based expert systems [17, 18, 28, 55] and attack graph-based approaches [34, 35, 39, 56, 58]. Several works have focused on languages for specifying attack signatures [15, 33, 41, 50] and state-transition analysis of anomalous system behavior [13, 25, 26],

An example of such system is STAT approach [26] that graphically models intrusions as transitions in a state machine. Each state in the state machine represents a snapshot of the monitored system as a set of assertions about the elements of the system. Each transition shows actions that move the system closer to the compromised state.

STAT can be considered as a high-level specification and, in that respect, compares with our SFT approach to modeling intrusions. A detailed representation of STAT state machine could be used as a design for an IDS or executed as an IDS, and in this respect, corresponds to the use of CPNs and agents in our system. However, the separate tools (SFTs, CPNs, and agents) used for the different concerns (requirements, design, and implementation) in our approach provide a clearer distinction between the development activities than it is done in an approach that uses state machines throughout the development lifecycle. Additionally, SFTs tend to be more understandable as a high-level specification than state machines.

As our work is based on integration of SFT and CPN for intrusion detection we will primarily focus on the graph-based approaches.

One of the earlier misuse detection models, *Intrusion Detection In Our Time (IDIOT)*, was developed by Kumar and Spafford [34, 35]. The system employs Colored Petri Nets to represent intrusion signatures, *patterns*. Although, as authors suggested, CPN is the most suitable technique for conditional matching of patterns, several modifications of CPN were made (elimination of concurrency, removal of local condition variables at transitions, addition of start and final states etc.) to make IDIOT model generic and applicable to any well defined input.

Our proposed IDS is also based on Colored Petri Nets, however the concept is applied to design specification rather than a direct execution of a CPN to allow the implementer to improve performance. In addition, we define a transformation from the CPNs to the implementation of the software agent intrusion detection system that preserves the CPN semantics. Another benefit of our model is its ability to operate in a distributed environment using an agent-based approach.

Another graph-based approach to misuse intrusion detection, called GrIDS, *the Graph-Based Intrusion Detection System* [58] was designed for distributed attacks against networks. It dynamically builds activity graphs describing network traffic by applying user-defined rules to audit data. Nodes in the graphs represent hosts or aggregations of hosts while edges represent network activity. Rather than building a single graph including all system activities, individual graphs are maintained by rule sets. Each rule set matches certain events from the network audit trail and either builds a new graph or adjusts an existing graph.

The model also allows intuitive aggregation of nodes and edges into reduced graphs which provides higher level of analysis and data sharing, resulting in a scalable design. Although, this system is built to detect security policy violations, it should be possible to extend the model to analyze for anomalies based on

selected objects and events.

While GrIDS considers only communication patterns between hosts, our modeling technique applies to all events in the monitored system. Also, rather than directly using the graphical model, a mobile agent intrusion detection system is developed using the CPN model as the design specification to improve performance and allow flexibility in implementation.

Similar to GrIDS approach, *the Adaptable Real-time Misuse Detection system (ARMD)* represents misuses as directed acyclic graphs (DAGs) [39]. Abstract events are represented by nodes in a graph and edges show the ordering of inter-event rules satisfied by the nodes. The intra-event rules determine the nodes chosen for the graph. The inter- and intra-event rules together define misuse signatures (named MuSigs). If a graph is built such that a sink node has an edge to it, an intrusion is detected.

Unlike the model used by GrIDS or by our approach that allows for aggregation through unification of tokens, MuSig graphs are not amenable to aggregation. Edges in a MuSig graph only mean that a predicate has been satisfied and they have no values or attributes that can be aggregated. At the same time, nodes in a MuSig graph correspond to specific events, which can be hard to aggregate in the absence of structured methods to aggregate the attributes associated with the events.

Finally, MuSig graphs can not be used for anomaly detection since by definition a MuSig graph detects a misuse intrusion. Thus, the GrIDS-style object/event model seems to be more powerful for general misuse and anomaly intrusion modeling. While our proposed approach allows for anomaly detection, it has another advantage of not requiring matching of graphs, as CPN graphs are mainly used for the design specification.

In recent years, several methods have been proposed to represent intrusion signatures through attack graphs which can be constructed from the alerts reported by intrusion detection system [42, 43, 45, 56]. These graphs precisely model attack paths in the network through nodes representing host vulnerabilities and edges showing connectivity between these hosts [56]. While attack graphs are exhaustive and precise, their manual construction is tedious and often error-prone. Recently, several projects have focused on automatic generation of such graphs [56, 59]. Another concern related to attack graphs is their scalability. While it became possible to build attack graphs for large networks using automatic tools, it is still quite difficult to manage their complexity. Several visualization techniques have been proposed to cope with this problem [43, 44]. Our approach also employs attack graphs, however, graph representation of intrusion is only required for design of the IDS rather than actual intrusion detection.

8 Discussion and Conclusions

This paper details the procedure by which a distributed, agent-based IDS was implemented from a SFT-based requirements and a CPN-based design. Intrusions are divided into temporal components which are modeled using SFT. Constraint nodes, specifying trust, temporal, and contextual relationships, are used to augment SFTs and restrict the combinations of events which define intrusions. Algorithmic approaches are used to create CPN templates from augmented SFT and agent implementations from the CPN templates. The result is an intrusion detection system to detect intrusions which were specified by the original requirements.

Dividing components of intrusions into temporal stages allows the development of CPNs that detect individual attacks[3]. Composition of the CPNs into a hierarchy models the correlation of individual attacks to detect complete intrusions. Future work may investigate how attacks may fit together into complete intrusions and determine how to further compose CPNs. For example, if detectors for individual attacks are developed, data mining techniques such as frequent episodes [36] may discover groups of attacks that occur in combination. A detector for the group of attacks could be made by composing the individual detectors together.

Constraint nodes were added to enable augmented SFT to model temporal, contextual, and trust relationships between events. Such information is necessary to distinguish actual intrusions from events that bear

[3] A number of highly-effective intrusions (e.g., CodeRed II and Nimda) are simple, scripted attacks that do not follow the distinct temporal stages. Simpler intrusion detection systems that match single events, such as SNORT [51], tend to be effective at detecting these intrusions.

similarity to intrusions and improve the false-positive rate of the implemented system.

An algorithm is used to convert augmented SFT intrusion specifications into CPN detector design templates. This conserves the relational constraints of the augmented SFT and preserves the logic of the SFT. Likewise, an algorithm is used to convert CPNs into agent implementation templates. The implementation preserves the properties of the CPN design while providing agents for use as a distributed intrusion detection system.

The augmented SFT, conversion from augmented SFT to CPN template, and the implementation of the IDS using the CPN design templates act together to preserve the correctness from requirements to implementation. The requirements engineer must refine the initial augmented SFT by adding constraints to specify the temporal, contextual, and trust relationships between events that take place as part of the intrusions. The designer must complete the CPN design by adding places to provide tokens to the CPN and refining the tokens so that they unify to satisfy the contextual constraints.

Our use of SFT with trust, temporal, and contextual constraints to model intrusions for a requirement specification has assisted the development of CPNs for intrusion detection. The use of CPNs to model intrusion detection system is novel. Likewise, agents can be used to implement intrusion detection systems. Our requirements to use augmented SFT, CPNs, and intrusion detection agents structures the development of an intrusion detection system into a repeatable and verifiable process.

Agents in our prototype intrusion detection system function as CPN places and transitions. Places are generally static agents which either act as a source of information or hold information until a transition requests it. Transition agents are the active components which accept tokens from places, act on or unify the information in the tokens, and pass the resulting tokens to other places. Viewing MAIDS agents and data as an implementation of a CPN has conveniently generalized the system and enabled further development. Transition agents are given a set of places to visit by the user interface. Future enhancement will enable the transition agents to self-direct their travels. Such capability in an agent could allow evasion of an attacker or faster response to important events.

We have implemented a prototype FTP bounce attack detector based on the CPNs detailed in this paper using agent technology based on our MAIDS implementation.

Future work will include investigation of the length of time tokens that should be kept in places. Since performance of the IDS degrades significantly as meaningless tokens accumulate, the current policy allows an uncollected token to expire after a fixed timeout. One possible extension is to allow a timeout to be specified in the Token constructor, making it possible to script delays into an attack to evade detection. This complicates the CPN model by adding work for the system designer (who would have to specify token lifetimes as part of the SFT). Furthermore, the development of an algorithm for token garbage collection should be explored to address the underlying issue of token lifetime management.

The augmented SFT and CPNs presented in this paper model misuse intrusion detection. Ongoing work is investigating the application of these techniques to anomaly intrusion detection. One of our first steps was modeling rules learned by a data mining algorithm for anomaly intrusion detection with CPNs [23]. We have created an algorithm to transform the learned rules into a CPN. Further work is required to develop an augmented SFT that describes this data mining technique and other techniques for anomaly detection, and then leads to a CPN model of anomaly detection.

Acknowledgement

We would like to thank Professor Robyn Lutz for her contributions to Section 3.

References

[1] In *Proc. Of the 9th ACM Conf. on Comp. and Comm. Security*, 2002.

[2] Network Associates. Nachi worm. Online, march 2004. http://vil.nai.com/vil/content/v_100559.htm.

[3] J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.

[4] E. Amoroso. *Intrusion Detection*. Intrusion.Net Books, Sparta, NJ, USA, 1999.

[5] D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Detecting unusual program behavior using the statistical component of the next-generation intrusion detection expert system (NIDES). Technical Report SRI-CSL-95-06, Stanford Research Institute Computer Science Laboratory, May 1995.

[6] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, Fort Washington, 1980.

[7] J. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. H. Spafford, and D. Zamboni. An architecture for intrusion detection using autonomous agents. Technical Report COAST TR 98-05, Purdue University Department of Computer Sciences, 1998.

[8] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *Proceedings, IEEE DARPA Information Survivability Conference and Exposition (DISCEX I)*, 2000.

[9] J. M. Bradshaw, editor. *An Introduction to Software Agents*. MIT Press, Cambridge, MA, USA, 1997.

[10] P. Cabena, P. Hadjinian, R. Stadler, J. Verhees, and A. Zanasi. *Discovering Data Mining: From Concept to Implementation*. Prentice-Hall PTR, Upper Saddle River, NJ, 1998.

[11] CPN group at University of Aarhus, Denmark. Design/CPN online. Online, 2000. http://www.daimi.au.dk/designCPN/.

[12] M. Crosbie and G. Spafford. Defending a computer system using autonomous agents. Technical Report 95-022, COAST Laboratory, Department of Computer Sciences, Purdue University, Apr. 1994.

[13] B. J. d'Auriol and K. Surapaneni. A state transition model case study for intrusion detection systems. In *Proc. of the 2004 International Conference on Security and Management (SAM'04)*, pages 186–192, 2004.

[14] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, Feb. 1987.

[15] B. V. Eric Totel and L. Mé. A language driven intrusion detection system for events and alerts correlation. In *Proceedings ot the 19th IFIP International Information Security Conference*, 2004.

[16] "Fyodor" <fyodor@dhp.com>. Nmap stealth port scanner for network security auditing. Online, 1999. http://www.insecure.org/nmap/.

[17] T. Garvey and T. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, 1991.

[18] N. Habra, B. L. Charlier, A. Mounji, and I. Mathieu. ASAX : Software architecture and rule- based language for universal audit trail analysis. In *European Symposium on Research in Computer Security (ESORICS)*, pages 435–450, 1992.

[19] G. Helmer. *Intelligent multi-agent system for intrusion detection and countermeasures*. PhD thesis, Iowa State University, Ames, IA, USA, Dec. 2000.

[20] G. Helmer, J. Wong, V. Honavar, and L. Miller. Automated discovery of concise predictive rules for intrusion detection. *Journal of Systems and Software*, 60(3):165–175, Mar. 2002.

[21] G. Helmer, J. Wong, V. Honavar, and L. Miller. Lightweight agents for intrusion detection. *Journal of Systems and Software*, 67(1), 2003.

[22] G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, and R. Lutz. A software fault tree approach to requirements analysis of an intrusion detection system. In *Proceedings, Symposium on Requirements Engineering for Information Security*, volume 7, pages 207–220. Springer, 2002.

[23] G. Helmer, J. S. K. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection. In *Proceedings, IEEE Information Technology Conference*, pages 121–124, Syracuse, NY, USA, Sept. 1998.

[24] IKV++ GmbH Informations und Kommunikationssysteme, Berlin, Germany. *Grasshopper User's Guide, Release 2.2*, 2001. `http://www.grasshopper.de/index.html`.

[25] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *SP '93: Proceedings of the 1993 IEEE Symposium on Security and Privacy*, page 16, 1993.

[26] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, Mar. 1995.

[27] W. Jansen, P. Mell, T. Karygiannis, and D. Marks. Applying mobile agents to intrusion detection and response. Technical Report Interim Report - 6416, National Institute of Standards and Technology, Oct. 1999.

[28] M. D. Jean-Philippe Pouzol. Formal specification of intrusion signatures and detection rules. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, 2002.

[29] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1992.

[30] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 3. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1997.

[31] S. P. Joglekar and S. R. Tate. Protomon: Embedded monitors for cryptographic protocol intrusion detection and prevention. *J. UCS*, 11(1):83–103, 2005.

[32] D. M. Kienzle and W. A. Wulf. A practical approach to security assessment. In *Proceedings of the 1997 workshop on New security paradigms*, pages 5–16, Langdale, Cumbria, United Kingdom, 1998.

[33] C. Kruegel and T. Toth. Distributed pattern detection for intrusion detection. In *Network and Distributed System Security Symposium Conference Proceedings: 2002*, 2002.

[34] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, IN, USA, Aug. 1995.

[35] S. Kumar and E. H. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, Baltimore, MD, USA, Oct. 1994.

[36] W. Lee, S. Stolfo, and K. Mok. Algorithms for Mining System Audit Data*, Data Retrieval and Data Mining*. Kluwer Academic Publishers, Boston, MA, USA, 1999. T. Y. Lin and N. Cercone, eds.

[37] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, USA, 1995.

[38] N. G. Leveson and J. L. Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, SE-13(3):386–397, Mar. 1987.

[39] J.-L. Lin, X. S. Wang, and S. Jajodia. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings, IEEE Computer Security Foundations Workshop*, pages 190–201, Rockport, MA, USA, June 1998.

[40] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (p-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, 1999.

[41] C. Michel and L. Me. ADeLe: an attack description language for knowledge-based intrustion detection. In *Sec '01: Proceedings of the 16th international conference on Information security: Trusted information*, pages 353–368, 2001.

[42] P. Ning, D. Xu, C. G. Healey, and R. A. S. Amant. Building attack scenarios through integration of complementary alert correlation methods. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04),*, pages 97–111, 2004.

[43] S. Noel, M. Jacobs, P. Kalapa, and S. Jajodia. Multiple coordinated views for network attack graphs. In *Proceedings of the Workshop on Visualization for Computer Security*, 2005.

[44] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118, 2004.

[45] S. Noel, E. Robertson, and S. Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *Proceedings of the 20th Annual Computer Security Applications Conference*, 2004.

[46] ObjectSpace, Inc., Dallas, TX. *ObjectSpace Voyager Core Technology User Guide*, 1999. Version 3.0.0.

[47] C. Phillips and L. P. Swiler. Proceedings of the 1998 workshop on new security paradigms. In *New Security Paradigms Workshop*, pages 71–79, Charlottesville, Virginia, United States, 1998.

[48] P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. The common intrusion detection framework architecture. Online, 1999. `http://www.gidos.org/drafts/architecture.txt`.

[49] N. J. Puketza, K. Zhang, M. Chung, B. Mukherjee, and R. A. Olsson. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering*, 22(10):719–729, Oct. 1996.

[50] M. F. Raihan and M. Zulkernine. Detecting intrusions specified in a software specification language. In *COMPSAC (1)*, pages 143–148, 2005.

[51] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the Thirteenth Systems Administration Conference (LISA 99)*, Seattle, WA, USA, Nov. 1999. USENIX.

[52] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[53] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.

[54] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, New York, 2000.

[55] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert systems in intrusion detection: A case study. In *Proceedings of the 11th National Computer Security Conference*, 1988.

[56] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[57] M. Slagell. The design and implementation of MAIDS (mobile agent intrusion detection system). Technical Report TR01-07, Iowa State University Department of Computer Science, Ames, IA, USA, 2001.

[58] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS-a graph based intrusion detection system for large networks. In *19th National Information Systems Security Conference Proceedings*, pages 361–370, Oct. 1996.

[59] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *DISCEXII Proceedings, DARPA's Information Survivability Conference and Exposition*, 2001.

[60] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.

[61] Y. Wang, S. R. Behera, W. Johnny, G. Helmer, V. Honavar, L. Miller, R. Lutz, and M. Slagell. Towards the automatic generation of mobile agents for distributed intrusion detection systems. *Accepted by Journal of Systems and Software in August 2004, available at sciencedirect.com*.

[62] J. Wong, G. Helmer, V. Naganathan, S. Polavarapu, V. Honavar, and L. Miller. SMART mobile agent facility. *Journal of Systems and Software*, 56(1):9–22, 2001.