Abstraction-based Intrusion Detection in Distributed Environments

Peng Ning
North Carolina State University
and
Sushil Jajodia and Xiaoyang Sean Wang
George Mason University

Abstraction is an important issue in intrusion detection, since it not only hides the difference between heterogeneous systems, but also allows generic intrusion detection models. However, abstraction is an error-prone process and is not well supported in current Intrusion Detection Systems (IDSs). This paper presents a hierarchical model to support attack specification and event abstraction in distributed intrusion detection. The model involves three concepts: *system view, signature*, and *view definition*. A system view provides an abstract interface of a particular type of information; defined on the instances of system views, a signature specifies certain distributed attacks or events to be monitored; a view definition is then used to derive information from the matches of a signature and presents it through a system view. With the three elements, the model provides a hierarchical framework for maintaining signatures, system views as well as event abstraction. As a benefit, the model allows generic signatures that can accommodate unknown variants of known attacks. Moreover, abstraction represented by a system view can be updated without changing either its specification or the signatures defined on the basis of it. This paper then presents a decentralized method for autonomous but cooperative component systems to detect distributed attacks specified by signatures. Specifically, a signature is decomposed into finer units called *detection tasks*, each of which represents the activity to be monitored on a component system. The component systems (involved in a signature) then perform the detection tasks cooperatively according to the "dependency" relationships among these tasks. An experimental system called *CARDS* has been implemented to test the feasibility of the proposed approach.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Cooperative information systems, heterogeneous systems, intrusion detection, misuse detection

---

Name: Peng Ning
Address: Department of Computer Science, North Carolina State University, Raleigh, NC 27695; email: ning@csc.ncsu.edu.
Name: Sushil Jajodia and Xiaoyang Sean Wang
Address: MSN4A4, Center for Secure Information Systems, George Mason University, 4400 University Drive, Fairfax, VA 22030-4444; emails: {jajodia, xywang}@gmu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

1. INTRODUCTION

Intrusion detection has been studied for at least two decades since the Anderson's report [2]. As a second line of defense for computer and network systems, intrusion detection systems (IDSs) have been deployed more and more widely along with intrusion prevention techniques such as password authentication and firewall. Research on intrusion detection is still rather active due to the fast change of information technology.

Intrusion detection techniques can be classified into two categories: *misuse detection* and *anomaly detection*. Misuse detection looks for signatures (i.e., the explicit patterns) of known attacks, and any matched activity is considered as an attack. Examples of misuse detection techniques include the State Transition Analysis Toolkit (STAT) [19] and Colored Petri Automata (CPA) [27]. Misuse detection can detect known attacks effectively, though it usually cannot accommodate unknown attacks. Anomaly detection models the subject's (e.g., a user or a program) behaviors, and any significant deviation from the normal behaviors is considered as the result of an attack. Examples of anomaly detection models include the statistical models used in NIDES (NIDES/STAT) [20] and HAYSTACK [46]. Anomaly detection has the potential to detect unknown attacks; however, it is not as effective as misuse detection for known attacks. In practice, both misuse detection and anomaly detection are often used as complementary components in IDSs (e.g., EMERALD [43], JiNao [57]).

Abstraction, as explained in Webster's New World Dictionary of American English [37], is the "formation of an idea, as of the qualities or properties of a thing, by mental separation from particular instances or material objects". Abstraction in intrusion detection is considered important, mainly because of two reasons. First, the systems being protected as well as the IDSs are usually heterogeneous. In particular, a distributed system often consists of different types of component systems such as Windows and UNIX machines. Abstraction thus becomes a necessary way to hide the difference between these component systems and allow intrusion detection in distributed systems. For example, IETF's Intrusion Detection Message Exchange Format (IDMEF) provides an abstract data model and common message format for different types of IDSs to understand each other [8]. Second, abstraction is often used to hide irrelevant details so that IDSs can avoid unnecessary complexity and focus on essential information. As an example, JiNao uses an Event Abstraction Module to convert low-level IP packets into abstract events (for Link State Advertisement) so that it can have more concise and generic representations of attacks [21; 57].

However, abstraction in intrusion detection has not been well supported. The most common way of abstraction is to use it as a preparation process, and the abstraction process is performed in ad hoc ways. Such approaches may cause some problems as the IDSs and our knowledge of intrusions evolve. For example, in the Mitnick attack described in [41], the attacker first launches a SYN flooding attack to prevent a trusted host from accepting incoming TCP connection requests (i.e., SYN packets), and then tries to connect to another host using the IP address and TCP port being flooded as source IP and source port. To detect such an attack in a misuse detection system, a human user may specify an attack signature which involves a SYN flooding event and the related TCP connection. However, this signature has a potential problem: The attacker may use other ways to disable one or all of the TCP ports of the trusted host and launch a similar attack, which is not covered by the signature. To cover such attacks, the user has to write one signature for each possible way in which the attacker can disable a TCP port of the trusted host. In addition, when new ways of disabling a TCP port are discovered, the user needs to input additional signatures into the system.

Certainly, an experienced user may choose a smarter way. Instead of using the SYN flooding event in the signature, he (or she) may abstract it to an event that can disable a TCP port on the trusted host. Such a signature is generic enough to accommodate variants of the aforementioned Mitnick attack. However, there is no framework to support such an attempt in existing systems. The user may, for example, write a program (e.g., the Event Abstraction Module in JiNao [21; 57]) to abstract all events (including SYN flooding attack) that can disable a TCP port. Unfortunately, when a new way of disabling a TCP port is discovered, the user will have to modify the program to reflect the new information. Such a task is certainly not a pleasant one for an administrator.

The above example suggests that (1) abstraction is an error-prone process and (2) there is not enough support for effective and efficient abstraction in current IDSs. To address these issues, we propose a hierarchical model for attack specification and event abstraction. Our model is extended from the misuse detection model proposed in ARMD [31; 32], which was developed to deal with the portability of misuse signatures for host-based IDSs.

There are three essential concepts in our model: *system view*, *(misuse) signature*, and *view definition*. A system view provides an abstract representation of a particular type of observable information, which includes an event schema and a set of predicates. Event schema specifies the attributes that describe the abstract events on a system view, while the predicates tell the relationship among system entities. For example, we may have a system view *TCPDOSAttacks* that represents TCP based denial of service (DOS) attacks, where the event schema specifies the IP address and port number being attacked and the predicates indicate the severity of the attack. Although there may be different types of such attacks, the system view *TCPDOSAttacks* can hide the difference between them and provide an abstract representation.

A signature is a distributed event pattern that represents a distributed attack on the instances of system views. With system views providing abstract views of the information, the signatures can represent the attacks in a generic way. For example, we may have a signature for the Mitnick attack based on the system view *TCPDOSAttacks* (among others). As a result, the signature will still be applicable even if the attacker uses other methods instead of SYN flooding to disable a TCP port of the trusted host.

View definition is the critical concept that helps us exceed the limitation of the previous approaches. A view definition is used to derive information from the matches of a signature and present it through a system view. With view definition, our model provides a hierarchical framework for event abstraction. For example, we may have a system view *IPPacket* that provides information for IP packet events. On the basis of *IPPacket*, we may first define signatures for various TCP based DOS attacks such as SYN flooding attacks and Ping of Death (see [25] for details of these attacks) and then abstract them as TCP based DOS attacks on *TCPDOSAttacks* using view definitions. As a result, detection of the above attacks will generate events on *TCPDOSAttacks*. The introduction of view definition provides a toolkit to unify attack specification and event abstraction in the same framework.

Having the three elements, our model allows a flexible and dynamic way of maintaining signatures and system views as well as event abstraction. For example, when we initially specify the system view *TCPDOSAttacks*, we may only have knowledge of some attacks such as SYN flooding and Ping of Death. Certainly, we can specify it with whatever we know and even describe signatures (e.g., signature for the Mitnick attack) using *TCP-*

DOSAttacks once it is specified. However, if new types of TCP based DOS attacks are later discovered, we do not need to change either the specification of the system view itself nor the signatures (e.g., the Mitnick attack) described on the basis of it. Instead, we only need to specify signatures and corresponding view definitions for the new discovery.

One important problem that we have to solve for our model is how to detect the specified attacks. The techniques proposed in ARMD [31] can be slightly extended to address host-based intrusion detection; however, new techniques are needed for distributed intrusion detection, since it requires distribution and coordination mechanisms, which is beyond ARMD.

Several distributed IDSs have been proposed to address similar issues. Early distributed IDSs for small scale systems usually have all the information sent to a central place (possibly after being filtered). For example, ASAX preprocesses the audit data in distributed component systems and sends necessary information to a central place for analysis [34; 35]. However, such a method does not scale well to large distributed systems, because the information collected from a large distributed system may exceed the capacity of any single system and the intrusion detection related messages will take the network bandwidth.

Later systems (e.g., EMERALD [43] and GrIDS [50]) adopt a hierarchical framework, which organizes IDSs into a fixed hierarchy and requires low-level IDSs send designated information to high-level IDSs. For example, EMERALD organizes IDSs for individual hosts under the IDSs for departments, which are under an IDS for the entire enterprise [43]. The hierarchical approach scales better than the aforementioned centralized approach. However, it is not always the most efficient way to detect distributed attacks. For example, if two IDSs that are far from each other in terms of the hierarchy are designated to detect a known distributed attack, the data sent by them may have to be forwarded several times (to higher-level IDSs) before they can be finally correlated. Indeed, the two IDSs can communicate more efficiently if they directly talk to each other. Thus, it is worth further research to seek more efficient alternatives to the hierarchical approach.

In this paper, we present an approach to organizing autonomous but cooperative component systems to detect distributed attacks. Our approach is based on the dependency among the distributed events in a signature. Unlike the hierarchical approach, our approach organizes the cooperative IDSs according to the intrinsic relationships between the distributed events involved in attacks, and, as a result, an IDS needs to send a piece of information to another IDS only when the information is essential for detecting the attacks. To examine the feasibility of our approach, we develop an architecture to support our approach and implement an experimental system named Coordinated Attacks Response & Detection System (CARDS).

The contribution of this paper is two-fold. First, we provide a framework for distributed attack specification and event abstraction by extending the original ARMD model. In this framework, abstraction is considered as an on-going process, and signatures and view definitions can be used to update the semantics of a system view without changing the its specification and the signatures specified on the basis of it. As a result, signatures in our model can potentially accommodate unknown variants of known attacks. Although the specification of attack signatures and the choice of right abstraction still partially depend on the user's skill, this framework provides guidance and alleviates the burden of writing and maintaining signatures. Second, we develop a decentralized approach to detecting distributed attacks specified using the revised model. By considering the intrinsic relationship between the distributed events in a signature, we decompose a signature into smaller units called *detection tasks* that can be executed in different IDSs; moreover, we develop a distributed algorithm to coordinate the detection tasks so that the IDSs can cooperatively detect the attacks.

The rest of this paper is organized as follows. The next section discusses the related work. Section 3 presents the basic model, which is extended from the misuse detection model in ARMD to accommodate new requirements in distributed environments. Section 4 further extends the basic model and presents the hierarchical model for attack specification and event abstraction. Section 5 presents a decentralized approach to detect distributed attacks specified by our model. Section 6 gives a heuristic approach to generating execution plans used by the decentralized detection. Section 7 briefly describes an experimental system named CARDS for the approaches proposed in this paper. Section 8 concludes this paper and points out some future research directions.

2. RELATED WORK

Our work falls into the research domain of detecting intrusions distributed over multiple systems, including distributed intrusion detection systems and other related techniques. A survey of the early work on intrusion detection is given in [36], and an excellent overview of the current intrusion detection techniques and related issues can be found in a recent book [3].

Early distributed intrusion detection systems collect audit data from distributed component systems but analyze them in a central place (e.g., DIDS [47], ISM [15], NADIR [17], NSTAT [24] and ASAX [34; 35]). Although audit data are usually reduced before being sent to the central analysis unit, the scalability of such systems is limited due to the centralized analysis.

Recent systems paid more attention to the scalability issue (e.g., EMERALD [43], GrIDS [50], AAFID [49], and CSM [56]). EMERALD adopts a recursive framework in which generic building blocks can be deployed in a highly distributed manner [43]. Both misuse detection and statistical anomaly detection are used in EMERALD. GrIDS aims at large distributed systems and performs intrusion detection by aggregating computer and network information into activity graphs which reveal the causal structure of network activity [50]. AAFID is a distributed intrusion detection platform, which consists of four types of components: agents, filters, transceivers and monitors [49]. These components can be organized in a tree structure, where child and parent components communicate with each other. AAFID emphasizes on the architecture aspect of distributed intrusion detection; detailed mechanism for performing distributed intrusion detection is not addressed. JiNao is an IDS that detects intrusions against network routing protocols [21; 57]. The current implementation of JiNao focuses on the OSPF (Open Shortest Path First) routing protocol. A distinguished feature of JiNao is that it can be integrated into existing network management systems. It is mentioned that JiNao can be used for distributed intrusion detection [21; 57]; however, no specific mechanisms have been provided for doing so.

In terms of the way of performing distributed intrusion detection, our approach differs from these systems as follows: While our approach decomposes and coordinates distributed event collection and analysis according to the intrinsic relationships between the distributed events, the aforementioned systems either have no specific way to coordinate different IDSs (e.g., JiNao [21; 57]), or rely on some predefined hierarchical organization, which is usually determined by administrative concerns (e.g., EMERALD [43], GrIDS [50]). Compared with the hierarchical approach, our approach has the advantage that the component IDSs can exchange necessary information without forwarding it along the hierarchy.

NetSTAT is an application of STAT [19] to network-based intrusion detection [54; 55]. Based on the attack scenarios and the network fact modeled as a hypergraph, NetSTAT automatically chooses places to probe network activities and applies the state transition analysis. Our approach is similar to NetSTAT in the sense that both approaches can decide what information needs to be collected in various places. However, our approach also differs from NetSTAT in the following ways. NetSTAT is specific to network-based intrusion detection, while our approach is generic to any kind of distributed intrusion detection. Moreover, NetSTAT collects the network events in a distributed way, but analyzes them in a central place. In contrast, our approach analyzes the distributed events in a decentralized way, that is, the events are analyzed as being collected in various places.

Event abstraction has long been recognized as an important issue in intrusion detection as well as many other areas. It can not only remove the irrelevant details, but also hide the difference between heterogeneous systems. Several systems have adopted components for event abstraction. ARMD explicitly brought up the concept of abstract system to provide the abstract representation of system information [31; 32]. AAFID introduces filters as a data abstraction layer for intrusion detection agents [49]. JiNao adopts an event abstraction module to transform low-level network activities into high-level events [21; 57]. (There are other examples; however, we do not enumerate them here.) Our approach in this paper further extends these ideas by providing a hierarchical framework for event abstraction. In particular, we view event abstraction as a dynamic process (instead of a preparation stage as in most of the previous approaches). The advantage is that we can update the semantics of event abstraction without changing either its specification or the signatures defined on the basis of it, and thus have generic signatures that can accommodate the variants of known attacks.

Several approaches have been proposed to represent known attacks. Among the earliest ones are the rule-based languages such as P-BEST [33] and RUSSEL [34; 35]. Later work includes the state transition analysis toolkit (STAT) [18; 19] and its extension POSTAT [16], the Colored PetriNet Automaton (CPA) [27; 28], and JiNao Finite State Machine (JFSM) [21; 57]. The general philosophy of these approaches is to make the representation mechanisms easy to use and yet be able to represent most of the known attacks (if not all). Our representation mechanism can be considered as a variation of STAT or CPA, which removes the explicit states from the attack patterns. A new feature of our method is that it allows events to be extracted from signatures so that attack (or event) patterns can be specified in a hierarchical way. Moreover, our representation mechanism gives an opportunity to decompose a distributed attack pattern into smaller units that can be executed in a distributed manner.

Common Intrusion Detection Framework (CIDF) is an effort that aims at enabling different intrusion detection and response (IDR) components to interoperate and share information and resources [23; 42]. CIDF views IDR systems as composed of four kinds components that communicate via message passing: Event Generators (E-boxes), Event Analyzers (A-boxes), Event Databases (D-boxes) and Response Units (R-boxes). A communication framework and a common intrusion specification language are provided to assist the interoperation among CIDF components [10; 22]. Several efforts have tried to improve CIDF components' ability to interoperate with each other: The Intrusion Detection Inter-component Adaptive Negotiation (IDIAN) protocol helps cooperating CIDF components to reach an agreement on each other's capabilities and needs [11]; MADAM ID uses CIDF to automatically get audit data, build models, and distribute signatures for novel attacks so that the gap between the discovery and the detection of new attacks can be reduced [29]; finally, the query facility for CIDF enables CIDF components to request specific information from each other [39; 40].

IETF's Intrusion Detection Working Group (IDWG) has been working on data formats and exchange procedures for sharing information among IDSs, response systems, and management systems. XML has been chosen to provide the common format and an Intrusion Detection Message Exchange Format (IDMEF) has been defined in an Internet draft [8]. IDWG uses the Blocks Extensible Exchange Protocol (BEEP) as the application protocol framework for exchanging intrusion detection messages between different systems [44]; an Intrusion Detection Exchange Protocol (IDXP) is specified as a BEEP profile [12], and a Tunnel profile is provided for different systems to exchange messages through firewalls [38].

We view CIDF, IDMEF (IDXP) and their extensions as complementary to ours. First, in terms of representation, CIDF, IDMEF (IDXP), and their extensions provide common message formats and exchange procedures for IDSs to interoperate and understand each other, while our work provides a framework for event abstraction as well as specification of known intrusion patterns. Second, neither CIDF nor IDWG provides any specific way to coordinate different IDSs (indeed, as standards, they try to avoid any specific mechanism). Though MADAM ID enables different IDSs to collaborate with each other, the collaboration is limited to collecting audit data for new attacks and distributing newly discovered signatures [29]. In contrast, our approach decomposes a signature for a distributed attack into smaller units, distributes these units to different IDSs, and coordinates these IDSs to detect the attack.

The Hummer project is intended to share information among different IDSs [14]. In particular, the relationships between different IDSs (e.g., peer, friend, manager/subordinate relationships) and policy issues (e.g., access control policy, cooperation policy) are studied, and a prototype system HummingBird was developed to address these issues. However, the Hummer project is to address the general data sharing issue; what information needs to be shared and how the information is used are out of its scope. In contrast, our decentralized detection approach addresses the issue of efficiently detecting specific attacks; it is able to specify what information is needed from each site and how the information is analyzed. Indeed, our decentralized detection approach can be combined with the Hummer system to fully take advantage of its data collection capability.

Our work is based on the host-based misuse detection system ARMD [31; 32]. This paper further extends the result in ARMD in several ways. First, the attack specification model is extended to allow the representation of attacks across multiple systems. In particular, our model assumes interval-based events rather than point-based events; thus, not only event records directly derived from audit trails but also compound events are accommodated. Moreover, the revised model adopts the notion of negative event and can take into account exceptional situations. Second, the revised model allows hierarchical specification of event patterns, which not only provides a way to model distributed attacks, but also a framework for automatic event abstraction. Finally, we develop a decentralized approach to detecting distributed attacks.

There are many other related work, such as various anomaly detection models (e.g., NIDES/STAT [20], HAYSTACK [46]), data mining approaches (e.g., JAM [30], ADAM

[4]), various tracing techniques (e.g., DECIDUOUS [6; 7], thumbprinting [51]), and embedded sensors [26]. We consider these techniques as complementary to ours presented in this paper.

3. THE BASIC MODEL

In this section, we present our basic attack specification model, which is extended from the model in ARMD [31; 32]. A critical concept of our model is system view; each system view provides an abstract interface for one particular type of information. Attack patterns, which we call misuse signatures, are then specified as distributed event patterns on the basis of system view instances.

System views reflect the notion of abstraction in our model. We consider abstraction as an on-going process instead of a preparation stage. That is, a system view can be extended without changing the signatures defined on the basis of its instances. A direct advantage is that the signatures defined in our model are generic and can accommodate new attacks to a certain degree.

In the rest of this section, we describe our basic model, i.e., how we represent system views and attacks. In the next section, we will present a hierarchical framework for attack specification as well as event abstraction based on the basic model.

3.1 System View and Event History

Intuitively, a system view provides an abstract representation of one particular type of information provided by a system. The system underlying a system view may be one single host, a network segment, or a distributed system consisting of several hosts.

Both event information and relationships among system entities are provided through a system view. Events represent what have happened or are happening in the system, while relationships among system entities represent the system state at certain times. For instance, the fact that two files are owned by the same user can be represented by a relationship *same_owner* between them.

The time when an event occurs is intrinsic to the event. In distributed environments, the intrusion detection related events are usually not instantaneous in terms of time. For example, a TCP connection could span several hours. To accommodate such events, we consider that each event has a duration and associate an interval-based timestamp with it. Notation-wise, each timestamp is denoted in the form of [*begin_time*, *end_time*], representing the starting and the ending points of the time interval, respectively.

Similar to ARMD [31; 32], we use dynamic predicates to represent the relationships among system entities. A dynamic predicate is a predicate with time as one of its arguments. For example, the dynamic predicate *same_owner*[*t*](*file_x*, *file_y*), which represents whether files *file_x* and *file_y* have the same owner, is True if and only if the owners of *file_x* and *file_y* are the same at time *t*. Note that “static”, or regular predicates are special cases of dynamic predicates.

The notion of system view is formally stated as follows.

Definition 1 A *system view* is a pair (*EvtSch*, *PredSet*), where *EvtSch* (called *event schema*) is a set of event attribute names, each with an associated domain of values, and *PredSet* is a set of dynamic predicate names. An *event e* on (*EvtSch*, *PredSet*) is a tuple on *EvtSch* with a timestamp [*begin_time*, *end_time*]. ☐

A system view serves as an interface of the information of interest. Though a system

event	*VictimIP*	*VictimPort*	*begin_time*	*end_time*
e_1	*10.0.0.1*	*80*	*19:37:01*	*19:43:05*
e_2	*10.0.0.2*	*23*	*19:38:15*	*19:38:15*
e_3	*10.0.0.255*	-1	*19:40:50*	*19:45:00*
e_4	*10.0.0.3*	-1	*19:44:15*	*19:44:15*
...	...	...	...	...

Fig. 1. An event history on the system view *TCPDOSAttacks*

view itself is fixed once defined, the information provided through it can be extended. For example, when we define a system view for TCP/IP based denial of service (DOS) attacks, we may abstract the system view from *Teardrop* and *Land* attacks (Please refer to [25] for the details of the attacks). However, we may later discover *SYN flooding* and *Ping Of Death* attacks, which are also TCP/IP based DOS attacks. Such newly discovered information can be directly provided through the existing system view without changing either the system view specification or the signatures already defined on the basis of its instances.

Note that *begin_time* and *end_time* are implicit attributes of the event schema, which collectively represent the timestamps of events on the system view. The information provided through a system view, including both event and state information, is formalized as an event history on the corresponding system view.

Definition 2 An *event history* on the system view (*EvtSch, PredSet*) consists of (1) a finite set of events $\{e_1, \ldots, e_m\}$ on (*EvtSch, PredSet*) and (2) an instantiation of the dynamic predicate names in *PredSet* such that for each p in *PredSet* and each time point t when an event occurs, p is instantiated as a regular predicate, denoted $p[t](x_1, \ldots, x_k)$ (i.e., for each instantiation of $x_1, \ldots, x_k$, $p[t](x_1, \ldots, x_k)$ gives True or False). □

Example 1 A network monitor that reports DOS attacks that disable one or all the TCP ports of a host may have a system view *TCPDOSAttacks* = (*EvtSch1*, $\emptyset$), where *EvtSch1* = $\{$*VictimIP, VictimPort*$\}$. Each DOS attack is reported as an event on (*EvtSch1*, $\emptyset$). The domain of *VictimIP* is the set of IP addresses, and the domain of *VictimPort* is the set of all TCP ports plus -1. *VictimPort* being -1 means that all TCP ports (of the host) are disabled. An event history on *TCPDOSAttacks* is shown in figure 1.

As we discussed earlier, *TCPDOSAttacks* may be defined when we only know, for example, *Teardrop* and *Land* attacks. When we later discover new types of DOS attacks, for example, *SYN flooding* attack, we can still reuse the previously specified *TCPDOSAttacks*.

As another example, a host may have a system view *LocalTCPConn* = (*EvtSch2, PredSet2*) for the TCP connections observed on the local host, where *EvtSch2* = $\{$*SrcIP, SrcPort, DstIP, DstPort*$\}$ and *PredSet2* = $\{$*LocalIP[t](var_IP), Trust[t](var_host)*$\}$. The domains of the attributes are clear from the names. The dynamic predicate *LocalIP[t](var_IP)* evaluates to True if and only if *var_IP* is an IP address belonging to the local host at time *t*, and the dynamic predicate *Trust[t](var_host)* evaluates to True if and only if *var_host* is trusted by the local host at time *t*. Examples of event history on *LocalTCPConn* are omitted. □

3.1.1 *Qualitative Temporal Relationships between Events.* The representation and reasoning about the qualitative temporal relationships between interval-based events have been extensively studied by the AI community [1; 13]. With these relationships, we can provide a more concise representation of the patterns among events.

Here we quote the thirteen relationships between intervals [1] and the eleven relation-

relation	*meaning*	*inverse relation*
e_1 `equal` e_2	$e_1.begin\_time = e_2.begin\_time$ *and* $e_1.end\_time = e_2.end\_time$	`equal`
e_1 `before` e_2	$e_1.end\_time < e_2.begin\_time$	`after`
e_1 `meets` e_2	$e_1.end\_time = e_2.begin\_time$	`inv-meets`
e_1 `overlaps` e_2	$e_1.begin\_time < e_2.begin\_time$ *and* $e_1.end\_time < e_2.end\_time$ *and* $e_1.end\_time > e_2.begin\_time$	`inv-overlaps`
e_1 `during` e_2	$e_1.begin\_time > e_2.begin\_time$ *and* $e_1.end\_time < e_2.end\_time$	`inv-during`
e_1 `starts` e_2	$e_1.begin\_time = e_2.begin\_time$ *and* $e_1.end\_time < e_2.end\_time$	`inv-starts`
e_1 `finishes` e_2	$e_1.begin\_time > e_2.begin\_time$ *and* $e_1.end\_time = e_2.end\_time$	`inv-finishes`
e_1 `older (than)` e_2	$e_1.begin\_time < e_2.begin\_time$	`younger (than)`
e_1 `head-to-head` e_2	$e_1.begin\_time = e_2.begin\_time$	`head-to-head`
e_1 `survives` e_2	$e_1.end\_time < e_2.end\_time$	`survived-by`
e_1 `tail-to-tail` e_2	$e_1.end\_time > e_2.end\_time$	`tail-to-tail`
e_1 `precedes` e_2	$e_1.end\_time <= e_2.begin\_time$	`succeeds`
e_1 `contemporary` e_2	$e_1.begin\_time < e_2.end\_time$ *and* $e_2.begin\_time < e_1.end\_time$	`contemporary`
e_1 `born-before-death` e_2	$e_1.begin\_time < e_2.end\_time$	`die-after-birth`

Fig. 2. The qualitative temporal relationships between two events

ships between semi-intervals [13] as the qualitative relationships between events. Figure 2 shows the relationships between two interval-based events e_1 and e_2. The inverse relation in the figure refers to the relation derived by switching the positions of the events in the original relation. For example, the inverse relation of e_1 `before` e_2 is e_1 `after` e_2, which is equivalent to e_2 `before` e_1.

Complex qualitative (temporal) relationships between two events can be represented by logical combinations of the above relations. For example, the situation that events e_1 and e_2 do not overlap in time can be represented by $(e_1$ `before` $e_2)$ *or* $(e_1$ `after` $e_2)$, or simply $e_1($`before` *or* `after`$)e_2$. In the following, we will take advantage of these qualitative temporal relationships to describe attack signatures.

3.2 Misuse Signatures

Misuse signatures are event patterns that represent intrusive activities across multiple systems. With system views as abstract representations of the underlying systems, a misuse signature is defined as a pattern of events on the instances of these system views. Specifically, a signature is a labeled directed graph. Each node in the graph corresponds to an *observable* event on a particular system view, and each labeled arc to a qualitative temporal relationship between the two nodes (events) involved in the arc. Events matched to the nodes must satisfy certain conditions, which are built into the model by associating a *timed condition* with each node (in a way similar to ARMD).

There are two kinds of events, *positive events* and *negative events*, due to their different "roles" in attacks. Positive events are the events that are necessary and critical for an attack. In other words, positive events are those necessary steps that an attacker cannot miss in order to launch the attack. Let us look at the Mitnick attack described in the introduction. In order to attack host *B*, the attacker first initiates a SYN flooding attack to prevent a TCP port of host *A*, which is trusted by *B*, from accepting any connection requests. (See [45] for detailed information about SYN flooding attack.) During the SYN flooding attack, the attacker tries to establish a TCP connection to *B* pretending (by IP spoofing) to be from the port being flooded. If the attacker succeeds, he can do whatever host *B* allows host *A* to do, since the attacking computer is mistaken for *A*. In this attack, the SYN flooding attack against host *A* and the TCP connection to host *B* from the attacking computer are positive events, because the attack will not succeed without them.

However, the existence of positive events does not always imply an attack. For example, even if we observe two positive events, a SYN flooding attack against host *A* in the network traffic and the corresponding TCP connection on host *B* during the SYN flooding attack, they do not constitute a Mitnick attack if the TCP connection is indeed initiated from host *A* (rather than from the attacking computer). In other words, if we also observe the same TCP connection on host *A*, then the TCP connection is just a normal connection during the SYN flooding attack rather than a part of the Mitnick attack. We call the TCP connection observed on host *A* a negative event, which serves as *counter evidence* of attacks. Thus, negative events are such events that if they coexist with the positive events, the positive events do not constitute an attack.

Negative events have appeared in different forms in other models. For example, CPA uses (negative) invariant to specify what *must not* happen during an attack, that is, the specified attack does not occur if the associated invariant is violated (matched) [27; 28]. Negative events are important to reduce false alarms; however, they should be used with cautions. The signature writer should be certain that the existence of negative events indeed indicates the non-existence of attacks. Otherwise, the attacker may bypass the IDS by intentionally creating negative events.

In order to model the patterns among multiple events, we use variables to help specify timed conditions. A variable is assigned an event attribute value from one node and then used in a timed condition associated with another node. We also use ε as a variable for an event. A timed condition is formally defined as follows.

Definition 3 A *timed condition* on a system view (*EvtSch, PredSet*) is a Boolean formula with atoms being either (1) comparisons between constants, variables, and event attribute names in *EvtSch*, or (2) of the form $p[\varepsilon.begin\_time](a_1, \ldots, a_k)$, $p[\varepsilon.end\_time](a_1, \ldots, a_k)$, $(\exists t \in [\varepsilon.begin\_time, \varepsilon.end\_time])\ p[t](a_1, \ldots, a_k)$, or $(\forall t \in [\varepsilon.begin\_time, \varepsilon.end\_time])$ $p[t](a_1, \ldots, a_k)$, where p is a dynamic predicate name in *PredSet* and $a_1, \ldots, a_k$ are constants, variables, or event attribute names in *EvtSch*. A timed condition evaluates to True or False when the variables are replaced with constants and ε with an event. □

We are now ready to formally define the concept of misuse signature.

Definition 4 Given a set of system view instances $S = \{(EvtSch_1,\ PredSet_1),\ \ldots,\ (EvtSch_k,\ PredSet_k)\}$, a *misuse signature* (or *signature*) on *S* is a 7-tuple (*N, E, SysView, Label, Assignment, TimedCondition, PositiveNodes*), where

(1) (*N, E*) is a directed graph,

(2) *SysView* is a mapping that maps each node *n* in *N* to a system view instance in *S*,

(3) *Label* is a mapping that maps each arc in *E* to a qualitative temporal relationship between two events,

(4) *Assignment* is a mapping that maps each node *n* in *N* to a set of assignments of event attributes in the system view *SysView*(n) to variables (denoted as *variable* := *at-*

tribute_name) such that each variable appears in exactly one assignment in the signature,

(5) *TimedCondition* is a mapping that maps each *n* in *N* to a timed condition on *SysView*(n) such that all variables in the timed condition appear in some assignments specified by (4), and

(6) *PositiveNodes* $\neq \emptyset$ is a subset of *N*. □

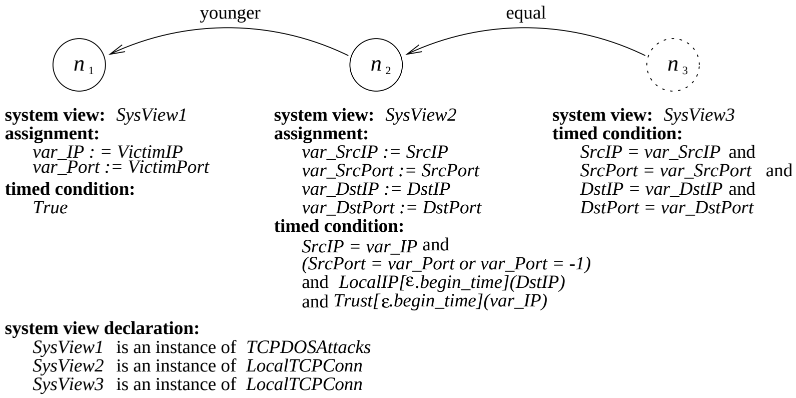
A misuse signature is an event pattern that represents an intrusive activity over a set of systems represented by the system view instances. The pattern is described by a set of events and the constraints that these events must satisfy. Given a signature (*N, E, SysView, Label, Assignment, TimedCondition, PositiveNodes*), the set *N* of nodes represents the set of events involved in the pattern, the edges and the labels associated with the edges, which are specified by the mappings *E* and *Label*, encode the qualitative temporal relationships between these events, the mapping *TimedCondition* specifies the conditions that each event must satisfy, and the mapping *Assignment* determines attributes that are used in some timed conditions. The set *PositiveNodes* of nodes represents the positive events necessary to constitute an attack, while (*N* − *PositiveNodes*) represents the negative events that contribute information to filter out false alarms.

Note that we use the qualitative temporal relationships between events to help specify misuse signatures. However, in order to represent quantitative temporal relationships between events, we will have to assign timestamps to variables and specify them in timed conditions. For example, if we require that two events e_1 and e_2 start within 10 seconds, we can assign $e_1.begin\_time$ to a variable t and then specify e_2's timed condition as $|t - e_2.begin\_time| < 10$ (assuming that the time is measured in second).

To have a better illustration, we pictorially represent a misuse signature as a labeled graph. Given a signature *Sig* = (*N, E, SysView, Label, Assignment, TimedCondition, PositiveNodes*), the components *N* and *E* are represented by a directed graph, where the nodes in *PositiveNodes* have solid boundary and the other nodes have dotted boundary, the components *SysView, Assignment* and *TimedCondition* are represented by a system view, a set of assignments and a timed condition associated with each node, the component *Label* is represented by a label associated with each arc, and the system view instances underlying the signature are given by a list of declarations of system view instances. An example of misuse signatures is shown as follows.

Example 2 This example shows a signature of the aforementioned Mitnick attack. It's worth noting that though the original attack involves a SYN flooding attack, an attacker can actually use other methods to disable one or all of the TCP ports of host *A* and achieve the same effect. Thus, the signature for the Mitnick attack should use something more abstract than SYN flooding attack.

Figure 3 shows a generic version of the signature for the Mitnick attack. The signature involves three system view instances: an instance of the system view *TCPDOSAttacks* and two instances of *LocalTCPConn*. (The system views *TCPDOSAttacks* and *LocalTCPConn* have been described in example 1.) This signature defines a generic pattern for the Mitnick attack. Node n_1 and n_2 represent positive events. Node n_1 represents a DOS attack on an instance of the system view *TCPDOSAttacks* (e.g., a network monitor), and node n_2 represents a local TCP connection event observed on one host, say host *B*. The timed condition associated with n_2 says that it is from the port being attacked (or any port if all TCP ports are disabled) and destined to host *B*, and the attacked host is trusted by *B*. The labeled arc (n_2, n_1) restricts that this TCP connection should occur after the begin time



system view: *SysView1*
assignment:
var_IP : = VictimIP
var_Port := VictimPort
timed condition:
True

system view: *SysView2*
assignment:
var_SrcIP := SrcIP
var_SrcPort := SrcPort
var_DstIP := DstIP
var_DstPort := DstPort
timed condition:
SrcIP = var_IP and
(SrcPort = var_Port or var_Port = -1)
and *LocalIP[ε.begin_time](DstIP)*
and *Trust[ε.begin_time](var_IP)*

system view: *SysView3*
timed condition:
SrcIP = var_SrcIP and
SrcPort = var_SrcPort and
DstIP = var_DstIP and
DstPort = var_DstPort

system view declaration:
SysView1 is an instance of *TCPDOSAttacks*
SysView2 is an instance of *LocalTCPConn*
SysView3 is an instance of *LocalTCPConn*

Fig. 3. The signature for the Mitnick attack

of the DOS attack. Representing a negative event, node n_3 stands for a TCP connection observed on the host being DOS attacked, say host *A*. The timed condition of n_3 and the labeled arc $(n_3,\ n_2)$ indicate that this TCP connection should be the same as the one represented by n_2. This signature says that a TCP connection from a port being DOS attacked (or any port of a host all TCP ports of which were disabled) is a Mitnick attack, if the host being DOS attacked does not have the same connection.

The signature in figure 3 reflects the basic idea of describing the signature for the Mitnick attack. It can also be revised to take into account that events n_1 and n_2 are often close to each other in time (in addition to that n_2 is `younger than` n_1). Such a signature would be easier to execute, since we do not have to consider a DOS attack and a TCP connection if they are far from each other in time. ☐

Now we clarify the semantics of a misuse signature by formally defining what it matches.

Definition 5 Let *Sig* = (*N*, *E*, *SysView*, *Label*, *Assignment*, *TimedCondition*, *PositiveNodes*) be a signature on the set of system view instances *S* = $\{(EvtSch_1,\ PredSet_1),\ \ldots,\ (EvtSch_k,\ PredSet_k)\}$, and for each $1 \leq i \leq k$, let $\mathcal{H}_i = \{e_{i,1}, \ldots, e_{i,m_i}\}$ be an event history on the system view $(EvtSch_i, PredSet_i)$. For a subset N_s of *N*, a mapping $\pi : N_s \rightarrow \{e_{1,1}, \ldots, e_{1,m_1}, e_{2,1}, \ldots, e_{2,m_2}, \ldots, e_{k,1}, \ldots, e_{k,m_k}\}$ is said to be a *match* of N_s on $\mathcal{H}_1, \ldots, \mathcal{H}_k$ if the following conditions are satisfied:

(1) for each node *n* in N_s, $\pi(n) = e$ is an event on the system view associated with *n* and event attribute values of e are assigned to variables according to the assignments associated with *n*,

(2) for each arc (n_1, n_2) in *E* such that n_1 and n_2 are in N_s, if $\pi(n_1) = e_i$ and $\pi(n_2) = e_j$ and (n_1, n_2) is mapped to a qualitative temporal relationship $\cdot R \cdot$ by the mapping *Label*, then $e_i \cdot R \cdot e_j$, and

(3) for each node n in N_s with a timed condition, if $\pi(n) = e$, then the timed condition is True with ε replaced with e and the variables with the values assigned in (1).

A match π of *PositiveNodes* on $\mathcal{H}_1, \ldots, \mathcal{H}_k$ is said to be a *match* of *Sig* if (1) *N* = *PositiveNodes*, or (2) $N \neq$ *PositiveNodes* and there does not exist a match π' of *N* such that π and π' are the same for nodes in *PositiveNodes*. ☐

Event#	*VictimIP*	*VictimPort*
e_{11}	*www.victim.com*	*80*
e_{12}	*www.victim.com*	*80*
e_{13}	*flooded.victim.com*	*513*

(a) Events generated by the network monitor

Event#	*SrcIP*	*SrcPort*	*DstIP*	*DstPort*
e_{21}	*one.victim.com*	*8765*	*target.victim.com*	*23*
e_{22}	*host.another.com*	*4000*	*target.victim.com*	*7*
e_{23}	*flooded.victim.com*	*513*	*target.victim.com*	*514*

(b) Local TCP events on the host *target.victim.com*

Event#	*SrcIP*	*SrcPort*	*DstIP*	*DstPort*	*Protocol*
e_{31}	*one.victim.com*	*8789*	*flooded.victim.com*	*23*	
e_{32}	*host.another.com*	*7863*	*flooded.victim.com*	*21*	
e_{33}	*flooded.victim.com*	*20*	*host.another.com*	*7864*	

(c) Local TCP events on the host *flooded.victim.com*

Fig. 4. Events on the system views

Example 3 Suppose the network monitor in example 2 has detected the DOS attacks shown in figure 4(a). (For simplicity, all timestamps are omitted.) Also, suppose the hosts *target.victim.com* and *flooded.victim.com* have TCP connection events shown in figure 4(b) and 4(c), and *target.victim.com* trusts *flooded.victim.com* (i.e., for all time point *t*, *Trust[t](flooded.victim.com)* = True). Suppose event e_{23} is `younger than` event e_{13} (i.e., $e_{13}.begin\_time < e_{23}.begin\_time$). Then e_{13} and e_{23} satisfy all the conditions specified for nodes n_1 and n_2. In addition, there does not exist any event on the host *flooded.victim.com* that satisfies the conditions for node n_3 along with e_{13} and e_{23}. Thus, events e_{13} and e_{23} constitute a match of the signature. In other words, an instance of the Mitnick attack is detected. □

Nodes in a signature represent events on system views; therefore, in the following discussion, we will use nodes and events interchangeably.

4. DERIVING SYSTEM VIEWS FROM SIGNATURES: A HIERARCHICAL MODEL

In this section, we extend the basic model described in section 3 to derive information from the matches of signatures and present it through (possibly existing) system views. Such a derivation provides a way to extract (or aggregate) information from the events that match the corresponding signatures, and thus provides a more concise view of what has happened or is happening in the systems. As a result, our model allows signatures to be specified hierarchically, since we can both describe signatures on the basis of system views and derive system views from signatures.

There are several benefits of this extension. First, it reduces the complexity of specifying signatures if the corresponding attacks can be decomposed into logical steps (or components). Having the ability to hierarchically define signatures allows a user to decompose a complex attack into logical components and resolve them separately, and thus allows a divide and conquer strategy in the signature specification process. Second, this approach provides a framework for (event) abstraction. A hierarchy of system views and signatures provides a way to abstract compound as well as aggregated information. Third, the framework is dynamic and flexible. With system views as the foundation of signatures, we can specify a signature on the basis of some system view instances and then later extend the system views without changing the specifications of either the system views or the signature. The derivation of an existing system view makes the abstraction represented by the system view a dynamic process.

4.1 View Definition

Intuitively, we derive the information on a system view (which is called the *derived system view*) from the matches of a signature in two steps. Step 1: for each combination of events that match the signature, we take the necessary information from them by assigning their attribute values to the variables (which is indicated by the assignments). In other words, the selected attribute values of an event are assigned to the variables if the event corresponds to a positive node in a match. As a result, we can consider that each signature has a relation whose attributes are the variables that appear in the assignments associated with positive nodes, and each tuple in this relation consists of the attribute values assigned to the variables in a match. We call the schema of such a relation the *matched view* and the information provided through the matched view the *match history* of the signature. For example, the signature of the Mitnick attack (shown in figure 3) has a matched view whose schema is (*var_IP, var_Port, var_SrcIP, var_SrcPort, var_DstIP, var_DstPort*), and the tuples in this view will be the corresponding IP addresses and port numbers involved in Mitnick attacks.

Step 2: we apply a function to process the matched history. The function takes the tuples on the matched view as input and outputs events on the derived system view. In particular, we identify a special class of functions that can be executed in real time. That is, the function can be applied to a match of the signature once it is detected. The events (on the derived system view) generated this way correspond to the compound events represented by the detection of signatures. For simplicity, we choose a subset of the dynamic predicates in the underlying system views as the predicate set in the new one. An easy extension could be to use logical combinations of the underlying dynamic predicates in the derived system view.

In the following, we first formally define the notions of matched view and matched history, then formalize the derivation of system views from signatures as *view definition*.

Definition 6 Let *Sig* = (*N, E, SysView, Label, Assignment, TimedCondition, PositiveNodes*) be a signature on a set of system view instances *S* = $\{(EvtSch_1, PredSet_1), \ldots, (EvtSch_k, PredSet_k)\}$. The *matched view* of *Sig* derived from S, denoted (*V*), is a relation schema that consists of all the variables appearing in the assignments associated with the nodes in *PositiveNodes*. Moreover, for each i, $1 \leq i \leq k$, let $\mathcal{H}_i$ be an event history on $(EvtSch_i, PredSet_i)$. Then the *matched history* of *Sig* derived from $\mathcal{H}_1, \ldots, \mathcal{H}_k$ is a relation on (*V*) that consists of one tuple *t* for each match of *Sig* on $\mathcal{H}_1, \ldots, \mathcal{H}_k$ and the attribute values of *t* are the values assigned to the variables in the match. ☐

Definition 7 Given a set *S* of system view instances, a *view definition* on *S* is a 4-tuple (*Sig*, *EvtSch, PredSet*, f), where

(1) *Sig* is a signature on *S*;

(2) *EvtSch* is a set of event attribute names, each with an associated domain of values;

(3) *PredSet* is a subset of all the dynamic predicates appearing in *S*;

(4) f is a function that takes tuples of the matched view of *Sig* and outputs tuples on *EvtSch* with interval-based timestamps.

The system view (*EvtSch*, *PredSet*) is called the system view derived by the view definition, or simply *derived system view*. ☐

A view definition (*Sig, EvtSch, PredSet*, f) derives a system view on the basis of the signature *Sig*. *EvtSch* specifies the event schema of the derived system view, *PredSet* indicates the dynamic predicates inherited from the underlying system views, and f describes how the matches of *Sig* are transformed into events on *EvtSch*.

A critical component of a view definition is the function f. A special class of function f is to post-process the matches of the signature and generate a compound event for each match. Typically, such a function f takes a match of *Sig*, selects some attributes of interest, and presents them through the derived system view. Information extracted this way may be used for high-level attack correlation or intrusion response. For simplicity, we use an SQL query of the form *SELECT-FROM-WHERE*, which is targeted to a single instance of the matched view, to specify such a function. Note that using a simplified SQL query does not imply that we have to use a SQL engine or DBMS; it can be simply executed by taking a match once it is detected, evaluating the condition in the *WHERE* clause and renaming the variables of interest. Such queries can be executed at the time when the matches of the signatures are detected, and thus support real-time processing of the detection results.

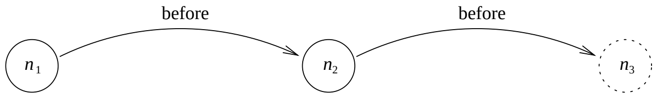
However, SQL queries (even in unrestricted forms) are not expressive enough; some event processing semantics cannot be expressed using such queries. For example, aggregation in terms of sliding time window cannot be expressed using SQL. An alternative approach is to use rule-based languages to describe the function f. Rule-based languages are more expressive than *SELECT-FROM-WHERE* SQL statements; however, they cannot cover all possible event processing semantics, either. For example, both P-BEST [33] and RUSSEL [34; 35] depend on external functions to extend their expressiveness. In addition, unlike *SELECT-FROM-WHERE* SQL statements, which can be executed by evaluating conditions and choosing/renaming attributes, rule-based languages require additional mechanisms to execute the rules.

Additional work is required to clarify what representation mechanisms are needed to specify the function f of a view definition. However, since our focus in this paper is the framework of signature specification and event abstraction, we consider f as a customizable blackbox and use *SELECT-FROM (V)-WHERE* for some special cases.

Example 4 Suppose we modify the signature *Mitnick* in figure 3 by associating additional assignments *var_tm1 := begin_time* and *var_tm2 := end_time* to nodes n_1 and n_2, respectively. We can have a view definition *MitnickAttacks*= (*Mitnick'*, {*Attack, VictimHost, VictimPort, TrustedHost*}, {*Trust[t](var_host)*}), where *Mitnick'* is the revised signature and *Query* is defined by the following SQL statement.

SELECT 'Mitnick' AS Attack, var_DstIP AS VictimHost, var_DstPort AS VictimPort,
var_SrcIP AS TrustedHost, var_tm1 AS begin_time, var_tm2 AS end_time
FROM (V)

The signature: *HalfOpenConn*



n_1:
system view: *SysView*
assignment:
var_SIP := SrcIP
var_SPort := SrcPort
var_DIP := DstIP
var_DPort := DstPort
var_Seq1 := SeqNum
var_time := begin_time
timed condition:
Flag = SYN

n_2:
system view: *SysView*
assignment:
var_Seq2 := SeqNum
timed condition:
SrcIP = var_DIP and
SrcPort = var_DPort and
DstIP = var_SIP and
DstPort = var_SPort and
AckNum = var_Seq1 + 1 and
(Flag = SYN/ACK or *Flag = RST)*

n_3:
system view: *SysView*
timed condition:
Flag = ACK and
SrcIP = var_SIP and
SrcPort = var_SPort and
DstIP = var_DIP and
DstPort = var_DPort and
AckNum = var_Seq2 + 1

system view declaration:
SysVew is an instance of *TCPPacket*

The view definition: *VD = (HalfOpenConn, {VictimIP, VictimPort}, {}, f),*
where *f* is a function that takes matches of the signature and output
the IP addresses and port numbers involved in SYN flooding attack.

Fig. 5. The view definition for deriving SYN flooding events from TCP packets

The system view derived by *MitnickAttacks* is ({*Attack, VictimHost, VictimPort, TrustedHost*}, {*Trust[t](var_host)*}). □

Example 5 We consider a view definition that aggregates low-level TCP/IP packet events into compound events representing SYN flooding attacks on the system view *TCPDOSAttacks* (discussed in example 1). Suppose the TCP/IP packet information is provided through a system view *TCPPacket* = (*EvtSch*, ∅), where *EvtSch* = {*SrcIP, SrcPort, DstIP, DstPort, SeqNum, AckNum, Flag*}. These attributes represent the source address, the source port, the destination address, the destination port, the sequence number, the acknowledge number and the flag associated with each packet. The domain of *Flag* is the set of valid TCP flags, including *SYN, SYN/ACK, ACK, RST*, etc. The domains of other attributes are clear from the names. (This system view can be directly generated by tools like tcpdump.)

Figure 5 shows the signature *HalfOpenConn* and the view definition *VD* that defines the derived system view on the basis of *HalfOpenConn*. Node n_1 represents a SYN packet that initiates a TCP connection, and node n_2 represents a SYN/ACK packet or a RESET packet that responds to the SYN packet. Both node n_1 and n_2 represent positive events. The negative node n_3 represents an ACK packet that finalize a TCP three-way handshake. Thus, a match of *HalfOpenConn* indicates either a half open connection or a connection reset during the TCP three-way handshake. The component f is a function (or procedure) that takes the matches of *HalfOpenConn* as input and output the IP address, TCP port and the timestamp of SYN flooding attacks. One way to implement f is to use a sliding time window and report a TCPDOSAttack (which is a SYN flooding attack in this case) if the half open connections against a certain TCP port exceeds a certain threshold. □

The reader may have noticed that some view definitions (e.g., the one shown in example 5) may introduce delays into the system. For instance, in example 5, to correctly generate interval-based events for SYN flooding attacks, the function f cannot output an event until the attack completes (for example, when the number of half open connections to a certain port drops below a certain threshold). There is a dilemma involved in such delays. On the one hand, derived events with correct timestamps are essential in reasoning about the

Attack	*VictimHost*	*VictimPort*	*TrustedHost*
Mitnick	*target.victim.com*	*514*	*flooded.victim.com*

Fig. 6. Events in the derived history

attacks, which implies that we may have to tolerate the delays. On the other hand, such delays have negative impact on intrusion detection: The system may miss the opportunity to respond to some attacks.

One possible way to alleviate this situation is to split each interval based event into a start event and a stop event. As a result, the attacks that only involve the starting point of long events can be detected more promptly. However, we only consider fully generated events in this paper, but view such an approach as a possible future work.

We call the information that a derived system view extracts or aggregates from the underlying event histories a derived event history, which is formally defined as follows.

Definition 8 Let *VD* = (*Sig, EvtSch, PredSet, f*) be a view definition on a set of system view instances *S* = $\{(EvtSch_1,\ PredSet_1),\ \ldots,\ (EvtSch_k,\ PredSet_k)\}$ and for $1 \le i \le k$, let $\mathcal{H}_i$ be an event history on $(EvtSch_i,\ PredSet_i)$. Then the *event history on* (*EvtSch, PredSet*) *derived from* $\mathcal{H}_1, \ldots, \mathcal{H}_k$ consists of

(1) a set of events that f outputs by taking as input the matched history of *Sig* derived from $\mathcal{H}_1, \ldots, \mathcal{H}_k$, and

(2) the instantiation (in $\mathcal{H}_1, \ldots, \mathcal{H}_k$) of the dynamic predicate names in *PredSet*.

Each event in the derived event history is called a *derived event*. □

Example 6 Consider the view definition shown in example 4. Suppose the event histories on the system view instances underlying this view definition are the same as in example 3 (see figure 4). Then based on the discussion in example 3, the events in the derived event history are shown in figure 6, and *Trust[t](var_host)* evaluates to True if *var_host* is trusted by the victim host at time *t*. □

The introduction of view definition allows a hierarchical organization of system views and signatures. Such a hierarchical model not only provides a framework for attack specification and event abstraction, but also ensures that abstraction becomes a dynamic and on-going process. Figure 7 shows a hierarchy of system views, signatures, and view definitions. Notice that such a hierarchy may evolve along time. For example, when we initially specify the system view *TCPDOSAttacks*, we may be only aware of two types of such attacks: *Land* and *Teardrop* (see [25] for details). Thus, we may derive information on *TCPDOSAttacks* from *IPPacket* using signatures for *Land* and *Teardrop* and the corresponding view definitions, as shown in the dashed box in figure 7. Certainly, signatures can be specified on the basis of *TCPDOSAttacks* once it is defined. However, we may later discover other TCP based DOS attacks, e.g., Ping of Death and SYN flooding attacks [25]. If this happens, we can use signatures and view definitions (e.g., the signatures and view definitions outside of the dashed box in figure 7) to derive more events on *TCPDOSAttacks* without changing either the specification of *TCPDOSAttacks* itself or the signatures defined on the basis of it. In other words, we can gradually change the semantics of system views and signatures without changing their specifications. As a result, the signatures specified in our model are generic and can potentially accommodate new attacks.

As illustrated in figure 7, an instance of system view may have multiple sources to derive

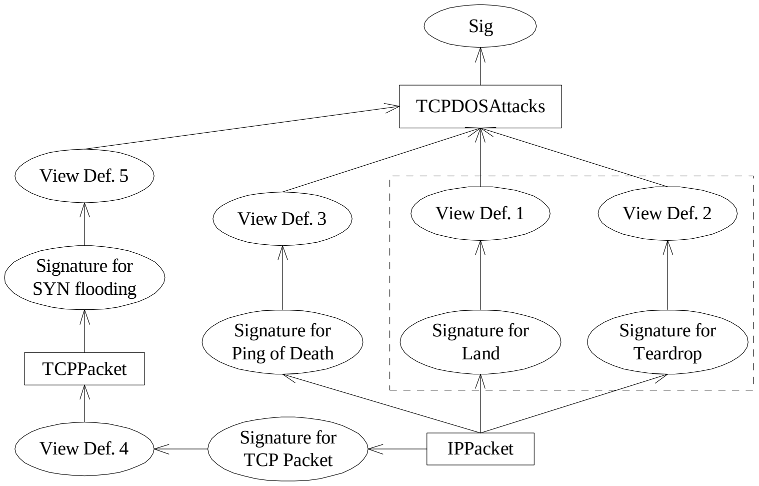


Fig. 7. A hierarchy of system views and signatures

its event history. For example, one event history on *TCPDOSAttacks* may be derived using all the four view definitions below it. To ensure the derived information is meaningful and consistent, we require that an event history on a derived system view must be generated from the same set of event histories on the underlying system views. In our example, all information of an event history on *TCPDOSAttacks* must be derived from the same event history on *IPPacket*.

4.2 Discussion

4.2.1 *Representable Attacks.* Our model is only applicable to the attacks that generate observable evidences. That is, the model can represent an attack only if the attack leaves characteristic traces that can be observed. (Such traces may be generated by scanning tools such as SAINT.) In addition, the model requires the user understand the attack well so that the user can write a signature according to his (or her) knowledge about the attack. In other words, our model, which shares the common drawback of other misuse detection models, is a tool that helps the reasoning about known attacks.

An attack can be represented as a signature as long as we can identify the critical events and their relationships that characterize the attack. Examples of such attacks include those that involve a sequence of events, partial ordered events, and concurrently occurring events.

Signature itself is not suitable to model the attacks with frequency of events as characteristics. For example, one may consider that there is a SYN flooding attack if and only if the number of half-open connections against a certain TCP port within a sliding time window exceeds a threshold T. We may specify a signature as having T half-open connections (events) that are destined to the same TCP port and that are all within the sliding time window; however, the signature will be awkward because of the large number of events resulting from a possibly large T. Fortunately, using view definition can help reduce the complexity of signatures. For example, we may simply specify a signature for the half-open connections and then use a view definition to count the half-open connections and extract a SYN flooding event (attack) when the count exceeds the threshold.

4.2.2 *Difficulty of Writing Signatures.* As noted in other models (e.g., CPA [27] and RUSSEL [34]), writing misuse patterns is not trivial. Our model is not an exception. In our model, writing misuse signatures requires a clear understanding of the critical evidences (i.e., distributed events) that indicate the existence of the attack. An incorrectly specified signature may give the attackers the opportunity to fool the system. For example, if the timed condition is a conjunction of several predicates and one of them is not essential, the attacker may try to make the predicate false so that he can bypass the detection. However, if the signature really reflects the exact nature of the attack, the attackers will not be able to launch an attack without triggering the signature, provided that they do not delete the evidences. In particular, negative events should be given special care, since incorrectly used negative events may give the attackers a chance to bypass the system.

Choosing the right system views is critical for having a signature that can accommodate variations of the attack. For example, to specify the signature of the Mitnick attack (Figure 3, example 2), we may simply use a system view that provides SYN flooding events for event n_1 (as in the original attack). However, the attacker may use another kind of denial of service attack to disable the TCP port of host *A* (or even host *A* itself) and avoid being matched by the signature. In contrast, if we use the system view *TCPDOSAttacks*, the attacks will always be accommodated as long as the attacker tries to disable a particular TCP port of host *A* (or host *A* itself). Therefore, choosing less abstracted system views may decrease the generality of the signatures.

4.2.3 *Clock Discrepancy.* The model assumes that all the clocks in different systems are well synchronized, which usually is not true in the real systems. The clock discrepancy can certainly affect the detection of attacks. For example, consider a signature that requires two events happen at the same time, it would be difficult to detect such an attack if the two events are from two different systems whose clocks do not agree.

A simple countermeasure is to set up a threshold t as the maximum difference between distributed clocks and handle timestamps in a special way. Specifically, two time points t_1 and t_2 in two different places are considered "equal" if $|t_1 - t_2| < t$, and t_1 is considered "before" t_2 if $t_1 - t < t_2$. A higher threshold will certainly help to tolerate worse clock discrepancy, but it will also result in a higher false alarm rate.

A true solution is to have a distributed time synchronization service that keeps the clock discrepancy between related systems at a tolerable level. This service should be secure so that attackers cannot create clock discrepancy to avoid being detected. Some solutions to secure time protocols have been given in [5; 52]. Though very interesting, this problem is out of the scope of this paper. Nevertheless, the clock discrepancy problem is not unique to our model; if the time relationship between distributed events is intrinsic to an attack, we cannot avoid this problem no matter what model we use.

5. DETECTING DISTRIBUTED ATTACKS

Detecting attacks specified by misuse signatures in a centralized way has been studied in ARMD [31; 32]. Although we have extended the model, the techniques developed in ARMD are still valid with slight changes. However, centralized detection of distributed attacks requires that all data be sent to one place, which is quite inefficient due to the communication overhead. Indeed, it is not a scalable solution; when the distributed system grows large, the data to be sent may exceed the network bandwidth and the processing capability of any single computer.

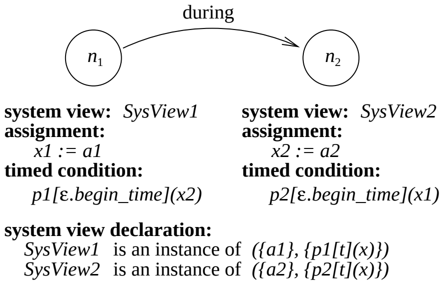


Fig. 8. A non-serializable signature

In this section, we explore the opportunities of detecting distributed attacks in a decentralized manner. We assume that the component systems trust each other and the communication between component systems is authenticated.

5.1 Generic and Specific Signatures

A signature in our model specifies a generic pattern of a certain type of attacks, which is usually independent of any specific systems. This is because signatures are defined on the abstract representations of underlying systems (i.e. system views), and this abstraction usually leads to generic signatures that can accommodate variants of the original attack. However, when the attack is to be detected, the signature has to be mapped to specific systems so that the IDS can reason the events observed on the specific systems according to the signature.

To distinguish the aforementioned two situations, we call a signature a *specific signature* if each system view instance used by the signature is associated with a system that provides information (i.e., event history) for the corresponding system view. In contrast, we call a signature a *generic signature* if there is at least one system view instance not associated with any system. For example, the signature shown in figure 3 is a generic one since no system view instance is associated with any system. If we associate the three system view instances to a network monitor *M* and two hosts *A* and *B*, respectively, it becomes a specific signature representing the Mitnick attack on these systems. Note that in this paper we consider whether a signature is generic or specific as a system feature, and do not include it in the formal model.

One generic signature usually corresponds to more than one specific signature, since the attacks modeled by the generic signature may happen against different targets. It is desirable to model attacks as generic signatures. When the attacks are to be detected for particular target systems, the specific signatures can be generated from the generic ones by associating the system view instances with appropriate systems. However, this does not mean that one can only write generic signatures. One can also write a specific signature for a particular system according to the configuration of the system.

5.2 Serializable Signatures

Not all signatures can be detected efficiently in a distributed way. Before presenting our approach for distributed misuse detection, we first demonstrate a “problematic” signature and then identify the signatures with which we have efficient algorithms.

Consider the signature shown in figure 8. Assume that the systems underlying the system view instances *SysView1* and *SysView2* are *host 1* and *host 2*, respectively. Note that the timed condition associated with n_1 needs the variable *x2* assigned at node n_2, while the timed condition with n_2 requires the variable *x1* assigned at n_1. Suppose both *host 1* and *host 2* want to process their events locally. Whenever an event e_1 occurs, *host 1* needs to send corresponding information (at least the value of *x1*) to *host 2*, since the value of *x1* is needed to evaluate the events represented by n_2. Similarly, all the events that occur on *host 2* need to be sent to *host 1*. In this example, each event requires a message. When there are more nodes that need information from each other, more messages will be required. For such signatures, a distributed approach entails more messages; indeed, a centralized approach is more appropriate.

The nature of this problem is the relationship between different nodes (events) in a signature. We clarify this relationship as follows. We say that a node n *requires* a variable x if x appears in the timed condition associated with n. For any two nodes n and n' in a signature, we say n *directly requires* n' if n requires some variables assigned at n'. Moreover, we say n *requires* n' if n directly requires n' or there exists another node n'' such that n requires n'' and n'' directly requires n'. For example, in figure 8 both n_1 requires n_2 and n_2 requires n_1, since n_1 requires the variable *x2* assigned at n_2 and n_2 requires the variable *x1* assigned at n_1. As another example, in figure 3 n_2 requires n_1, and n_3 requires both n_2 and n_1. Intuitively, node n needs information from node n' through variable assignments if n (directly) requires n'.

Now we identify the signatures that can avoid the above situations by the notion of serializable signature.

Definition 9 A signature *S* is *serializable* if (1) the binary relation *require* on the set of nodes in *S* is irreflexive, and (2) no positive node requires any negative node. ☐

The signature in figure 3 (on page 12) is an example of serializable signatures: the relation *require* on the set of nodes $\{n_1, n_2, n_3\}$ is irreflexive and the only negative node n_3 is not required by any node. However, the signature in figure 8 is not serializable, since n_1 and n_2 require each other.

Since the relation *require* is transitive by definition, it is implied that the relation *require* on the set of nodes of a serializable signature is a strict partial order.

Some non-serializable signatures can be transformed into equivalent serializable ones. Consider a variant of the signature in figure 3 in which the comparison $SrcIP = var\_IP$ is not placed in the timed condition associated with node n_2 but represented equivalently as $var\_SrcIP = VictimIP$ and placed in the timed condition associated with n_1. Then this signature is not serializable, since node n_1 and n_2 require each other. However, we can always transform it back into the equivalent form shown in figure 3. Indeed, for any two nodes that require each other in a non-serializable signature, if one of the problematic variables appears in a conjunctive term of the timed condition that does not involve dynamic predicate, we can always place it into another timed condition as we did above so that the two nodes no longer require each other.

In the rest of this paper, we will only consider the detection of attacks specified by serializable signatures.

5.3 Detection Task and Workflow Tree

A signature for a distributed attack is composed of events distributed over multiple systems. When detecting such attacks, communication between different systems is inevitable, since the evidences from different places need to be correlated in some way in order to reason about the attacks. To avoid transmitting all the distributed events to a centralized place for analysis, we let each system process the events that it observes and all the systems that are involved in a signature collaborate to perform the detection.

We consider the nodes in a signature as the basic processing units. An alternative is to treat all or some of the nodes in one system as one unit; however, we will have to process the events for different nodes in one unit differently and thus have a similar result. For the sake of description, we informally call the processing for a node n a *detection task* for n. (We will clarify the semantics of detection task later by a formal definition.) Intuitively, a detection task for node n determines whether an event corresponding to n satisfies the conditions related to n.

The detection tasks in a signature are not isolated due to the relationships between the nodes. In particular, the detection task for node n needs the information from the detection task for node n' if n requires n'. For example, consider the signature shown in figure 3. Given only the events on *SysView2*, the detection task for node n_2 will not be able to determine whether such an event satisfies the condition for n_2 without the variable var_IP assigned at node n_1. Therefore, we need to coordinate the detection tasks in a signature in order to successfully perform the intrusion detection.

Several issues are worth considering. First, the relation *require* on the set of nodes in the signature should be reflected in the coordination. As we discussed earlier, the relation *require* imposes that the detection task for node n' should give information to the detection task for n if n requires n'. Second, positive events represent possible attacks; to ensure the security of the system, positive events should be processed as soon as possible. Third, since the goal is to determine whether a set of events constitutes an attack, the results of all the detection tasks should finally be correlated together.

We use *workflow tree* to represent the coordination of the detection tasks in a signature. The nodes in a workflow tree consists of all the nodes in the signature, and an edge from one node to the other indicates that the detection task for the latter node should send information (variable values and timestamps) to the task for the former one. The workflow tree is formally defined as follows.

Definition 10 A *workflow tree* for a serializable signature *Sig* is a tree whose nodes are all the nodes in *Sig* and whose edges satisfy the following conditions: (1) given two nodes n_1 and n_2 in *Sig*, n_2 is a descendant of n_1 if n_1 requires n_2, and (2) there exists a subtree that contains all and only the positive nodes in *Sig*. ☐

Condition 1 says that the detection task for node n_1 (directly or indirectly) receives information from the detection task for node n_2 if n_1 requires n_2; condition 2 says that the detection tasks for positive events must be performed before any the detection tasks for any negative event. Moreover, the tree structure ensures that all the results of the detection tasks will finally be correlated together. Figure 9(a) shows a workflow tree for the signature in figure 3.

In a workflow tree, the root of the subtree that contains all and only the detection tasks for the positive nodes is called the *positive root*, and the root of the entire tree, if a negative node, is called the *negative root*. For example, in figure 9(a), node n_2 is the positive root while node n_3 is the negative root.

Note that a workflow tree specifies the coordination of the detection tasks involved in an attack; it does not specify the order of involved events. One node being a child of another

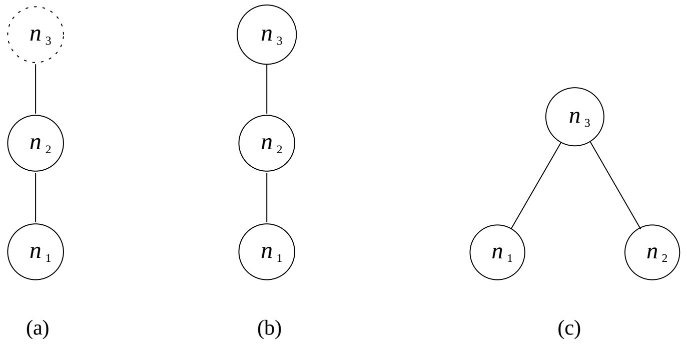


Fig. 9. Examples of workflow trees

does not imply that the former node must happen before the latter one.

Another issue is the arrangement of negative nodes in workflow trees. In our current approach, all positive nodes are arranged in a subtree, and no negative node can appear under positive nodes. One may think that having negative nodes between or under positive nodes can improve the performance. It is worth pointing out that negative events are possibly observable events, not filtering conditions. It is true that if we already observed some negative events, we may be able to invalidate some positive events without going through all positive nodes. However, such an arrangement also introduces difficult situations if we do not observe the corresponding negative events. In this case, we have to send the corresponding positive information to the rest of positive nodes, and two situations may follow. On the one hand, if we do not later discover the rest of positive events, then sending positive information through negative detection tasks already costs more. On the other hand, if we do find all positive events involved in a possible attack, we still need to check with the previously visited negative events, since some negative events may be observed after the previous check. Thus, the aforementioned approach does not always result in better performance than workflow tree. Further considering the simplicity of workflow tree, we choose our current approach to coordinate the detection tasks.

There are other alternative ways to represent the coordination of detection tasks. For example, we may represent the coordination of detection tasks in one signature as a directed acyclic graph with one sink, where the nodes are the detection tasks, and an edge from one detection task to another represents that the former detection task should send information to the latter one. An important distinction between this alternative representation and the workflow tree is that in the former representation, each detection task may send messages to multiple detection tasks, while in the workflow tree, each detection task sends information to at most one detection task, and the information required by more than one detection task is first sent to one task and then forwarded to the others. Since most of the events in normal situations are normal, most of the information sent by a detection task is expected not related to an attack. Using the workflow tree can not only simplify the model, but also reduce the number of messages when most of the messages are determined useless and not forwarded to other detection tasks. In some special cases where we would like to use signatures to, for example, assess the severity of an alert, all information sent by a detection task may contribute to the final assessment; however, we believe such signatures will not be the majority of the signatures we will use.

One signature may have multiple workflow trees. Consider a signature that has three positive nodes n_1, n_2 and n_3, where n_3 requires both n_1 and n_2. Figure 9(b) and 9(c) show two different workflow trees for this signature. We will discuss how to select appropriate workflow trees in the next section.

In the rest of this section, we first clarify what are detection tasks for a signature with a given workflow tree, and then describe how the detection tasks are executed.

Definition 11 Given a serializable signature *Sig* = (*N, E, SysView, Label, Assignment, TimedCondition, PositiveNodes*) and a workflow tree *T* for *Sig*, the *detection task* for each *n* in *N* is a 11-tuple (*n, sysview, assign, cond, p, PM, C, CPM, type, isRoot, negativeRoot*), where

- *sysview = SysView(n),*
- *assign = Assignment(n)* $\cup \{begin\_time_n := begin\_time, end\_time_n := end\_time\}$,
- *cond* is the conjunction of *TimedCondition(n)* and the qualitative temporal relationships (represented by *E* and *Label*) between *n* and *n*'s descendants in *T*,
- *p* is the parent node of *n* in *T* if *n* is not the root, or *p* is the positive root if *n* is the negative root,
- *PM* = $\{begin\_time_{n'}, end\_time_{n'} \mid n' = n$ or *n*'s descendants$\} \cup \{$the variables assigned at *n* and *n*'s descendants and required by *n*'s ancestors$\}$,
- *C* is the set of child nodes of *n* in *T* if *n* is not a leaf, and *C* contains the negative root node if *n* is the positive root node but not the root of the whole workflow tree,
- *CPM* is a mapping from *C* to sets of variables such that for each *c* ∈ *C, CPM(c)* is the *PM* component of the detection task for *c*,
- *type = positive* if *n* ∈ *PositiveNodes*, and *type = negative* otherwise,
- *isRoot = True* if *n* is the root of *T* or the subtree of *T* that consists of all the nodes in *PositiveNodes*, and *isRoot = False* otherwise, and
- if *n* is the positive root and *T* has negative nodes, *negativeRoot* is the root of *T*; otherwise, *negativeRoot* is *NULL* (invalid). □

The formal definition of detection task specifies the information required for the processing of each event in a signature as well as how this information is derived from the signature and the corresponding workflow tree. The component *n* identifies the event in the original signature; the component *sysview* indicates the system view instance from which the detection task gets event and state information about the monitored system; the component *cond* specifies the condition that should be tested against each event on *sysview*; the component *assign* includes all the assignments of event attributes to variables that should be performed if an event on *sysview* satisfies *cond*; the component *p* identifies the detection task (i.e., the *parent detection task*) to which this detection task should send detection related messages; the component *PM* specifies the schema of the messages to be sent to the parent detection task; the component *C* identifies all the detection tasks (i.e., the *child detection tasks*) from which this detection task is going to receive detection related messages; similar to *PM*, the component *CPM* specifies the schema of the messages to be received from each child detection task; the component *type* indicates whether this detection task corresponds to a positive or a negative event; finally, the component *isRoot* tells if this detection task corresponds to the root of the workflow tree or the subtree consisting of all the positive nodes.

Note that timestamp information can be implicitly represented in signatures, while in detection tasks, we explicitly represent the processing of timestamp information by timestamp variables (e.g., $begin\_time_n$ and end_time_n). This is because signatures are provided by human users; thus, using qualitative temporal relationships can usually make this job easier. However, detection tasks are developed for programs; hence, representing the timestamp information explicitly is required.

Let us see an example before we discuss the execution of detection tasks.

Example 7 Consider the signature for the Mitnick attack in figure 3. The detection task for n_1 is DT_1= (n_1, *SysView1*, $assign_1$, $cond_1$, n_2, PM_1, $\emptyset$, *NULL*, *positive*, *False*, *NULL*), where

- $assign_1$ = {*var_IP* := *VictimIP*, *var_Port* := *VictimPort*, $begin\_time_{n_1}$:= $begin\_time$, $end\_time_{n_1}$:= end_time},
- $cond_1$ is *True*, and
- PM_1 = {$begin\_time_{n_1}$, $end\_time_{n_1}$, *var_IP*, *var_Port*}.

This detection task says that it takes events on *SysView1*, which is an instance of *TCPDOSAttacks*, and for each event that satisfies $cond_1$, it makes the assignments in $assign_1$ and sends the assigned values to the detection task for n_2 as a tuple on PM_1.

The detection task for n_2 is DT_2 = (n_2, *SysView2*, $assign_2$, $cond_2$, n_3, PM_2, C_2, CPM_2, *positive*, *True*, n_3), where

- $assign_2$ = {*var_SrcIP* := *SrcIP*, *var_SrcPort* := *SrcPort*, *var_DstIP* := *DstIP*, *var_DstPort* := *DstPort*, $begin\_time_{n_2}$:= $begin\_time$, $end\_time_{n_2}$:= end_time},
- $cond_2$ is (($begin\_time > begin\_time_{n_1}$) *and* (*SrcIP* = *var_IP*) *and* (*SrcPort*=*var_Port*) *and LocalIP*[$\varepsilon.begin\_time$](*DstIP*) and *Trust*[$\varepsilon.begin\_time$](*var_IP*)),
- PM_2 = {*var_SrcIP*, *var_SrcPort*, *var_DstIP*, *var_DstPort*, $begin\_time_{n_1}$, $end\_time_{n_1}$, $begin\_time_{n_2}$, $end\_time_{n_2}$},
- C= {n_1, n_3}, and
- $CPM_2(n_1) = PM_1$, $CPM_2(n_3) = PM_3$.

This detection task says that it takes events on *SysView2*, which is an instance of *LocalTCPConn*, and receives tuples on PM_1 and PM_3 from the detection tasks for n_1 and n_3, respectively. For each event and the variables that satisfy $cond_2$, it makes the assignments in $assign_2$ and sends the assigned values to the detection task for n_3 as a tuple on PM_2. Note that n_2 is the positive root in the workflow tree. As we will see later, it processes the information from the detection task for the negative root n_3 in a different way.

The detection task for n_3 is DT_3 = (n_3, *SysView3*, $assign_3$, $cond_3$, n_2, PM_3, C_3, CPM_3, *negative*, *True*, *NULL*), where

- $assign_3$ = {$begin\_time_{n_3}$:= $begin\_time$, $end\_time_{n_3}$:= end_time},
- $cond_3$ is (($begin\_time = begin\_time_{n_2}$) *and* ($end\_time = end\_time_{n_2}$) *and* (*SrcIP* = *var_SrcIP*) *and* (*SrcPort* = *var_SrcPort*) *and* (*DstIP* = *var_DstIP*) *and* (*DstPort* = *var_DstPort*)),
- PM_3 = {$begin\_time_{n_1}$, $end\_time_{n_1}$, $begin\_time_{n_2}$, $end\_time_{n_2}$, $begin\_time_{n_3}$, $end\_time_{n_3}$},

```
Algorithm DetectionTask
Input: (1) A detection task DT = (n, sysview, assign, cond, p, PM, C, CPM, type, isRoot),
       (2) events on sysview, events from the detection tasks for the nodes in C, and events from the
           detection task for the negative root node if $n$ is the positive root.
Output: (1) partial match events sent to the parent task if $p \neq NULL$, and
        (2) all the matches of the signature if $n$ is the positive root.
Method:
    Let $h$ be an empty relation whose schema is $H = \{begin\_time, end\_time\} \cup \{$all the attributes of
    $sysview$ that appear in $cond$ or $assign\}$. For each $c \in C$, let $pmt_c$ be an empty relation whose
    schema is $CPM(c)$. If $n$ is the positive root, let $m$ be an empty relation whose schema is $PM$.

    for each event $e$ do
        Let $PMT := \emptyset$.
        if $e$ is an event on $sysview$ then
            Let $PMT := \Pi_{PM}(\rho_{(v:=a)\in assign}(a \rightarrow v, \sigma_{cond}(e \bowtie (\bowtie_{c \in C} pmt_c))))$.
            if $C \neq \emptyset$ then
                Let $h := h \cup \Pi_H(e)$.
        if $e$ is from a child detection task for $c \in C$ then
            Let $PMT :=$
                $\Pi_{PM}(\rho_{(v:=a)\in assign}(a \rightarrow v, \sigma_{cond}(e \bowtie h \bowtie (\bowtie_{c' \in C \wedge c' \neq c} pmt_{c'}))))$.
            Let $pmt_c := pmt_c \cup \Pi_{CPM(c)}(e)$.
        if type = positive and isRoot = True and $e$ is from the negative root then
            Let $m := m - \Pi_{PM}(e \bowtie m)$.
        if $PMT \neq \emptyset$ then
            if $p \neq NULL$ then
                Send each tuple in $PMT$ to the detection task for $p$ as an event.
            if type = positive and isRoot = True then
                Let $m := m \cup PMT$.
    end
```

Fig. 10. The algorithm for executing detection task

- $C = \{n_2\}$, and
- $CPM_3(n_2) = PM_2$.

This detection task says that it takes events on *SysView3*, which is also an instance of *LocalTCPConn*, and revceives tuples on PM_2 from its child detection task identified by n_2. For each event and the assigned variable values that satisfy $cond_3$, it makes the assignments in $assign_3$ and sends the assigned values to the detection task for n_2 as a tuple on PM_3 (since n_2 is the positive root in the workflow tree). Here we assume that each event from a particular system is uniquely identified by its timestamp. Note that node n_3 represents a negative event, and a satisfaction of $cond_3$ means that counter evidence is found for a previously discovered match of the signature. Thus, the detection task DT_2 can use the transmitted variable values to mark the match as a false alarm. ☐

5.4 Execution of Detection Tasks

The workflow tree provides a framework for the coordination of the detection tasks. In this subsection, we further explain how each detection task is performed in this framework. We assume that the processing of each event is atomic to ensure the correctness. Allowing concurrent processing of multiple events is interesting and may improve the performance; however, we do not cover it in this paper but consider it as possible future work.

Figure 10 shows the algorithm for executing a detection task. In order to have a concise description of the algorithm, we adopted some notations from the relational algebra. (Please refer to any database textbook (e.g. [53]) for the semantics of the notations.)

The algorithm uses several relations (tables) to keep events and detection result. The relation h, which we call the *history table*, keeps the necessary information of the events on the system view instance *sysview*. The attributes of the history table consist of the timestamps and the event attributes that appear in *assign* or the condition *cond*. For example, the detection task DT_1 in example 7 has the history table whose attributes consist of *VictimIP*, *VictimPort, begin_time* and *end_time*.

For each child node $c \in C$ of n, the relation pmt_c, which we call the *partial match table for* c, keeps the variable values assigned by the child detection task for c. The attributes of pmt_c include all the variables in $CPM(c)$. For example, for the detection task DT_2 in example 7, the partial match table pmt_{n_1} has attributes $begin\_time_{n_1}$, $end\_time_{n_1}$, *var_IP*, and *var_Port*.

If n is the positive root node in the corresponding workflow tree, the detection task then keeps a relation m (called the *matched table*) for the detection result of the signature. In figure 10, the attributes of the matched table include all the variables in *PM*. Alternatively, we can use all the timestamp variables in *PM* to identify the matches, assuming that each event is uniquely identified by its timestamp.

Note that it is necessary for a detection task to keep both the event information and the variable values received from its child tasks. The detection task may be able to determine that a previously examined event is involved in an attack after receiving additional information from its child tasks. Similarly, the detection task need to examine the information previously received from the child tasks when a local event occurs. Thus, each detection task needs to maintain both a history table and a partial match table for each child detection task. As an exception, the detection tasks without child tasks do not need to maintain any table.

The execution of the algorithm *DetectionTask* is presented in an event-driven fashion. For the sake of presentation, we also consider the variable values sent by a child detection task as an event whose attributes are the variable names. To distinguish between different events, we call the events on system view instances *raw events* and the events sent by a child detection task *partial match events*.

When the detection task receives a raw event e on the system view *sysview* (line 3), it first checks whether e satisfies *cond* along with some tuples in the partial match tables pmt_c. This is specified by a join of e with the partial match tables pmt_c for all c in C (line 4). The schema of the resulting relation is then changed to PM by a series of renaming operations followed by a projection (line 4). Then selected attribute values of e are saved in the history table h if necessary (lines 5 and 6).

When the detection task receives a partial match event e from a child task c (line 7), it first checks whether this event satisfies *cond* along with the historical events in h and partial match tables for c' other than c. Similarly, this is specified by a join of e with h and all $pmt_{c'}$ for all c' in C other than c (line 8). The schema of the resulting relation is then changed to PM by a series of renaming operations followed by a projection (line 8). Then selected attribute values of e are saved in the partial match table pmt_c as a tuple on $CPM(c)$ (line 9).

If the detection task generates new partial match events (i.e., $PMT \neq \emptyset$), it will send all the events to its parent detection task (denoted by p) if it has one (lines 12 to 14).

As a special case, if the detection task is for a positive root and there are negative nodes in the corresponding workflow tree, the discovery of partial matches (i.e., $PMT \neq \emptyset$) implies possible attacks represented by the signature. This seems to introduce a dilemma. On the one hand, we cannot conclude that an attack really happens if there are negative events in the signature, since we may later discover counter evidences that invalidate these attacks. On the other hand, if there really are attacks, not responding immediately may cause further damage to the systems.

However, from the perspective of the system security, avoiding damage to the system is more important than avoiding false alarms. Thus, we design the algorithm to work as follows. When the detection task for a positive root node discovers a non-empty PMT, it saves the result into the matched table m (lines 15 and 16), assuming they all correspond to attacks. When the detection task for the positive root receives an event e from the detection task for the negative root (i.e., counter evidence of previously detected matches), it then removes all matches that share the same attribute values as e (lines 10 and 11).

Note that the detection task for a negative event works in the exactly same way as those for positive events, unless the node is the root of the workflow tree. In the latter case, it sends to the detection task for the positive root the timestamps of the events involved in each partial match.

The execution of detection task can be quite complex if the detection task has many child tasks. The dominant steps are steps 4 and 8: Step 4 involves a conditional join of the most recently discovered event and the partial match tables, and step 8 involves a conditional join of the most recently received partial match event, the history table and the other partial match tables. A naive implementation of the join operation consists of testing the condition on all combinations of the new event and the tuples in the other tables. Thus, step 4 would involve $\Pi_{c \in C} |pmt_c|$ tests of the condition, and similarly, step 8 would involve $|h| \times \Pi_{c' \in C \wedge c' \neq c} |pmt_c|$ condition tests. Such a method corresponds to exhaustive search, and should be avoided in an implementation.

Two approaches can be used to reduce the execution cost. First, in-memory database query optimization techniques such as in-memory hybrid hash join [9] and T-Tree [48] can greatly reduce the cost of the join operation. Indeed, an in-memory database such as TimesTen [48] can be used in a component IDS to reduce the development cost. Second, some join operations may be materialized to speed-up the event processing. For example, we can pre-compute $\bowtie_{c \in C} pmt_c$ so that when a raw event is discovered, it can be directly joined with the pre-computed table. More research is needed to make the execution of a single detection task efficient; however, we do not address this problem in this paper but consider it as future work.

Assume that the communication between detection tasks is *resilient* (i.e., a message inserted into the communication channel will be delivered to the receipent eventually) and the clocks of different IDSs are well synchronized. The correctness of the decomposition and algorithm *DetectionTask* is assured by the following theorem.

THEOREM 1. *If all the detection tasks for a signature execute according to the algorithm* DetectionTask*, then they will detect all and only the matches of the signature.*

Three issues are worth further clarifying about Theorem 1. First, Theorem 1 ensures the detection of the matches of the signatures, not the attacks themselves. In other words, Theorem 1 says that under the aforementioned assumptions, we can detect the attacks as long as the evidences of the attacks are present in the event histories. Second, this theorem does not imply that each alarm raised by the system represents an attack, since an alarm may later be disabled due to the discovery of negative events. Instead, it says that each alarm represents an attack if it stays long enough. The exact threshold to decide whether an alarm is a true positive depends on the network delays and the processing time of the related detection tasks. However, we do not need to identify such a threshold, since it does not contribute to the operation of the system. Third, there is a hidden assumption under the theorem: infinite memory. That is, we never discard any information from the history tables or the partial match tables. However, in reality, we may have to discard some data from these tables due to the memory limitation. As a result, we may miss some stealthy attacks or have false alarms.

5.5 Optimization

As we pointed out earlier, one signature may have several workflow trees. Detecting the same signature with different workflow trees may result in different performance, since they may have different storage requirements and different patterns of message transmission between detection tasks. We will discuss how to select a good workflow tree in the next section.

Even with a fixed workflow tree, optimization is still available to improve the performance. Sometimes a detection task can determine that an event on the system view is not involved in an attack even if it needs information from other detection tasks. For example, in the workflow tree in figure 9(a), which is a workflow tree for the signature in figure 3, the detection task for n_2 can decide that a TCP connection event e is not involved in any attack if the destination IP address does not belong to the host being monitored (i.e., $LocalIP[e.begin\_time](DstIP) = False$). In general, we can transform the *cond* component of a detection task into the conjunction of two conditions such that one of them involves only the event attributes and constants. We call this part of the *cond* component a *sieving condition*. In the above example, the sieving condition is $LocalIP[e.begin\_time](DstIP)$. The sieving conditions can be evaluated without any information from other detection tasks, and only the events that satisfy the sieving condition need further consideration. Moreover, if we can measure the likelihood that each conjunctive term of *cond* is false, we can check those that have higher likelihood of being false and thus avoid unnecessary evaluations.

Another observation reveals further optimization opportunity. This can be explained with the above example as well. In order for an event e on the system view instance *SysView2* to be a part of an attack, it must satisfy $SrcIP = var\_IP$, which is implied by the condition $cond_2$. This means that we can replace the variable var_IP with the attribute $SrcIP$ in the predicate $Trust[\varepsilon.begin\_time](var\_IP)$ and still have an equivalent condition. As a result, the sieving condition of the detection task n_2 can be expanded to ($Trust[\varepsilon.begin\_time](Src\_IP)$ *and* $LocalIP[\varepsilon.begin\_time](DstIP)$) and filter out more events than the original one. In general, we can perform an equality analysis through the assignments and the timed conditions in the signature to find out all variables and attributes that are equivalent to each other. Then in the condition component *cond* of each detection task, if a variable is equivalent to an attribute of the system view instance, we can replace the variable with the attribute.

The reader may have noticed that when the size of the history and the partial match tables grow very large, the performance of the algorithm *DetectionTask* may be greatly affected due to the join operations (lines 4 and 7 in figure 10). Moreover, the detection task may not be able to grow these tables when memory is limited. A practical solution, which has been used many times in similar contexts (e.g., the token replacement policy in [27]), is to periodically remove out-of-date tuples from the history and the partial match tables (using certain replacement policy). This can certainly improve the performance; however, in theory the revised algorithm may miss some stealthy attacks, since some evidence could be removed before being correlated with others. More severely, an attacker may intentionally create a large amount of partial matches to launch a denial of service attack against the IDS. To detect such attempts, a statistic (e.g., the size of the table) may have to be associated with each history/partial match table. Nevertheless, this is a problem common to all IDSs that need to keep state about partial detection results (e.g., USTAT [19]).

The algorithm *DetectionTask* requires that each detection task maintain a history table for the raw events that are possibly involved in attacks. However, one event may be stored in more than one history table if several detection tasks take raw events from the same system view instance. An alternative way to avoid this situation is to maintain one history table for each system view instance. However, this approach is not a silver bullet either. When a detection task receives partial match events from its child tasks, it will have to determine whether a candidate event stored in the history table satisfies the condition together with the newly received information, which means that it will have to scan the records in the history table. Maintaining one history table for each system view instance will inevitably increase the size of the history table and thus increase the scanning time. A trade-off between time and space may be desirable, but is out of the scope of this paper.

6. GENERATING WORKFLOW TREE

As we discussed in section 5, different workflow trees for a given signature may result in different performance. Three major factors that reflect the performance are the CPU time, the message transmission and the storage requirement. The less requirement for the three factors, the better performance the workflow tree can result in.

Theoretically, we can define the optimal workflow tree for a specific signature as the one with the least CPU usage, message transmission and space requirement, and use the optimal workflow tree to gain the best performance. However, this requires a measurement of the CPU usage, the message transmission and the space requirement by the specific signatures, which involves not only how message are transmitted but also how often each type of events in the specific signature occurs and how conditions are evaluated.

An interesting approach to achieve this is to develop a cost model that can estimate the aforementioned measurements and select the optimal workflow tree according to the model. However, the development of the cost model will inevitably involve calculating (or estimating) the frequency of various types of events, which is usually time-consuming, and the resulting model will depend on the systems that are used to generate the cost model and may change as time goes on.

In the following, we present an alternative approach that considers several heuristics that usually lead to “good” workflow trees.

6.1 A Heuristic Approach

In this subsection, we discuss three principles for developing effective workflow trees as well as their relationships, and then present the heuristic algorithm that generates workflow trees on the basis of the principles.

The edges in a workflow tree represent the required information flows in the detection process. An edge implies message transmission between two systems if the detection tasks for the two nodes involved in the edge are located at different systems. If one node requires another node, a path between them is certainly unavoidable. However, unnecessary edges may result in unnecessary message transmissions and additional storage.

Consider a specific signature with three nodes n_1, n_2, and n_3, where n_3 requires both node n_1 and n_2, and the system views associated with the three nodes belong to three different systems. This signature has two workflow trees shown in figures 9(b) and 9(c). The workflow tree in figure 9(b) is bad, since the variable values assigned at n_1 must be sent to n_3 through n_2, resulting in two message transmissions and additional storage at n_2. On the other hand, the workflow tree in figure 9(c) is better than the previous one, since it allows the variable values assigned at n_1 and n_2 to be sent to n_3 separately. This example suggests that a workflow tree should avoid unnecessary edges. Principle 1 states this observation.

PRINCIPLE 1. *Place an edge from node n_1 to node n_2 only when (1) n_1 requires n_2, or (2) n_1 requires node n' and there is a path from n_2 to n'.*

If the inclusion of the edges is unavoidable, the system views associated with the two nodes involved in an edge should at least be located at the component system so that the necessary information flow will be processed in the same local system instead of being transmitted between different systems. Principle 2 states this observation.

PRINCIPLE 2. *If there has to be a path between two nodes that belong to the same system, place an edge directly between them whenever possible.*

Different events usually occur at different rates. We call the events that happen infrequently the *rare events*. For example, the event represented by node n_1 in figure 3, which is a DOS attack, is a rare event, since such an event seldom happens in a normal network. Note that rare events are relative and context dependent. A type of events that is rare in one situation may not be rare in another.

Rare events can help improve workflow tree generation. Note that a partial match event is generated at node n only if n has received partial match events from n's child nodes. Since rare events occur infrequently, the nodes that have rare events as descendants tend to generate fewer partial match events than the other nodes. Indeed, the lower the the placement of the rare events in the workflow tree, the fewer partial match events will occur, and the less message transmission and storage are required. This leads to principle 3.

PRINCIPLE 3. *Place the nodes for rare events as close to the leaves as possible.*

There may be conflicts between the principles. For example, the workflow tree shown in figure 9(c) is better than the one in figure 9(b) according to principle 1. However, if event n_1 is extremely rare, it *may* be better to choose the latter one according to principle 3. To precisely decide which one is better requires the cost of both workflow trees, which is what we wanted to avoid as discussed earlier.

Here we compare the three principles in normal situations and order them in terms of their priorities. First, let us compare principles 1 and 3 using the aforementioned example. Suppose n_1 is rare. At first glance, it seems that the workflow tree in figure 9(b) is much better than the one in figure 9(c). One reason could be that with the workflow tree in figure 9(b), detection task n_2 does not need to send messages to n_3 if it has not received anything from n_1. Moreover, if the history table of n_2 is limited to a certain size, some information

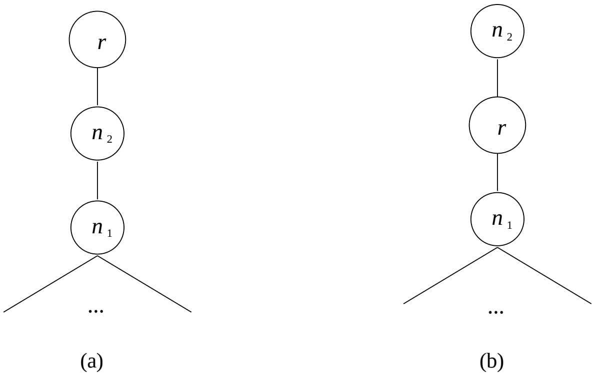


Fig. 11. Comparison of two workflow trees

may be dropped without being transmitted at all. However, this reasoning is flawed if we look at another aspect of this workflow tree. Note that n_2 does not require n_1, which implies that the information sent by detection task n_1 will not reduce the information stored in the history table of n_2. As a result, if we use the workflow tree in figure 9(b), each event detected by detection task n_1 will generate a partial match along with each tuple stored in the history table of n_2. The information sent to detection task n_3 is then the Cartesian product of the history tables of n_1 and n_2. In contrast, if the workflow tree in figure 9(c) is used, only the history tables need to be sent. Therefore, we assign principle 1 higher priority than principle 3. (Note that in extreme cases, i.e., when n_1 is *extremely* rare, the workflow tree in figure 9(b) may have less cost.) A similar reasoning can show that we should give higher priority to principle 1 than principle 2.

Now let us compare principles 2 and 3. Suppose r, n_1 and n_2 are three events (among others) involved in a signature. Assume that event r is rare, and events n_2 and n_3 are in the same component system. Figure 11(a) shows the workflow tree that we will select if principle 2 is given higher priority, and figure 11(b) shows the workflow tree that we will choose if we favor principle 3 over principle 2. With the workflow tree in figure 11(b), both n_1 and r need to transmit messages through the network in order to send information to their parent detection tasks. However, since event r is rare and it requires variables from its descendants, r will have much less partial matches than n_1, and the messages from r to n_2 will be much less than those from n_1 to r. In contrast, with the workflow tree in figure 11(a), only the messages from n_2 to r needs to be sent in the network, since n_1 and n_2 are located in the same component system. If we assume that sending messages from n_1 to r in the first case has roughly the same cost as transmitting messages from n_2 to r in the second case, then the workflow tree in figure 11(b) has a little more cost than the one in figure 11(a) in terms of message transmission. However, when detecting an attack, the workflow tree in figure 11(b) involves two messages between the component systems in which the three nodes are located, while the workflow tree in figure 11(a) only has one. Thus, the workflow tree in figure 11(b) introduces longer delay than the one in figure 11(a). Nevertheless, the workflow tree in figure 11(b) requires less space than the one in figure 11(a), since the partial match table of r in figure 11(b) will be smaller than that of n_2 in

```
Algorithm Gen_WFT
Input: A specific signature S and a subset $Rare$ of the nodes in S,
Output: A workflow tree $T$ for S.
Method:
    1. Let $result$ := an empty tree, $N$ = the set of positive nodes in S;
    2. Let $result$ := $Gen\_Tree(S, result, N, Rare)$;
    3. Let $N$ := the set of negative nodes in S;
    4. Let $result$ := $Gen\_Tree(S, result, N, Rare)$;
    5. return $result$.
```

```
Subroutine Gen_Tree
Input: A specific signature S, a tree $T_{in}$, a subset $N$ and a subset $Rare$ of the nodes in S.
Output: A tree $T_{out}$.
Method:
    1. Let $R$ := $\{T_{in}\}$;
    2. while $N \neq \emptyset$ do
    3.     Let $C$ be the set of nodes in $N$ that do not require any node in $N$;
    4.     for each node $n'$ in $C$, let its weight be the number of the trees in $R$ that have nodes
           required by $n'$ and whose roots are in the same component system as $n'$;
    5.     let $C'$ be the set of nodes in $C$ that have the largest weight;
    6.     if $C' \cap Rare \neq \emptyset$, let $C' := C' \cap Rare$;
    7.     let $n$ be any node in $C'$;
    8.     Let $N := N - \{n\}$, $R := R \cup \{n\}$;
    9.     for each tree $T$ in $R$ that has nodes required by $n$ do
    10.        Add an edge from $n$ to $T$'s root;
    11. if $|R| > 1$ (i.e. $R$ has more than one tree)
    12.    Group the trees in $R$ in such a way that the trees whose roots belong to the same
           component system are in one group;
    13.    Choose a group $G$ that has the maximum number of trees;
    14.    if there is a tree $T$ in $G$ that has no common nodes with $Rare$, let $r$ := $T$,
           else let $r$ be any tree in $G$;
    15.    for each tree $T \neq r$ in $R$, add an edge from $r$'s root to $T$'s root; let $result$ be the
           newly constructed tree;
    16. return $result$.
```

Fig. 12. The algorithm to generate a workflow tree from a specific signature

figure 11(a) due to the rareness of r. Here we care more about the speed of detection than the space requirement; thus, we decide to give principle 2 a higher priority. Principle 3 may have higher priority than principle 2 if space becomes a compelling concern.

Figure 12 shows the algorithm *Gen_WFT* developed according to these principles. The input of the algorithm consists of a specific signature *S* and a set *Rare* of the nodes corresponding to rare events, which is a subset of the nodes in *S*. The output of the algorithm is a workflow tree for *S*. The algorithm uses a subroutine *Gen_Tree* to help construct the target workflow tree. The algorithm *Gen_WFT* first generates the part of the workflow tree for the positive nodes and then the whole tree, ensuring that the processing of a positive event does not require the processing of any negative one(s).

In the subroutine *Gen_Tree*, the algorithm first constructs a set of trees according to the relationship *require*, the location where the events are observed, and the rare event information (steps 2 - 10). The algorithm always chooses a node that does not require any node not in R as the next candidate to be processed. Since the relation *require* is a strict partial order, the algorithm can always find a candidate event.

As shown in steps 4 - 7, the algorithm tries to find the nodes that will arrange the most

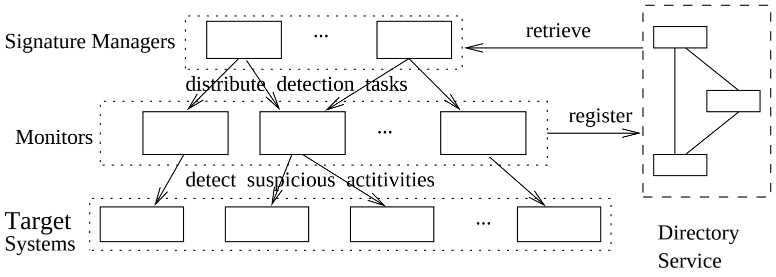


Fig. 13. The CARDS architecture

number of nodes in one component system (principle 2), and then tries to find a rare event from them (principle 3). In steps 9 and 10, the algorithm adds edges only when necessary (principle 1).

After processing all of the input nodes, the subroutine *Gen_Tree* builds one single tree if the previous steps result in more than one tree (steps 11 - 15). The algorithm chooses one of them and adds edges from its root to the roots of all the others. To minimize the message transmission, the algorithm chooses the tree whose root belongs to the component system having the most roots of the aforementioned trees (principle 2). Again, the algorithm places rare events as close to the leaves as possible by trying not to choose a tree that has nodes representing rare events (principle 3).

7. CARDS: AN EXPERIMENTAL SYSTEM

A prototype system named Coordinated Attack Response and Detection Systems (CARDS) is being developed to explore the feasibility of the approaches proposed in this paper. The preliminary design and development of CARDS has been described in [58]. CARDS is composed of three kinds of independent but cooperative components: *signature manager*, *monitor* and *directory service*. Figure 13 shows the architecture of CARDS. In a typical environment, there may be one or more signature managers and one or more monitors. The monitors can be embedded in the monitored system or as a dedicated system separate from the monitored system. Different monitors can cooperate with each other through message passing when they are involved in cooperative detection of attacks.

As shown in figure 13, with the monitor configuration information (i.e., what system view instances are provided by the monitors) retrieved from the directory service, a signature manager generates specific signatures from generic signatures, decomposes specific signatures into detection tasks (according to the methods in sections 5 and 6), and distributes these tasks to the involved monitors.

Given a generic signature, the signature manager generates specific signatures for all possible systems that can provide the system view instances required by the generic signature. This is because in theory the attack specified by the generic signature may happen to all these systems; thus, all of them should be protected against the attack. However, people may not want to detect all specific signatures of a generic one due to some administrative concerns. We envisage some support for the generation of specific signatures according to different administrative concerns. However, this is out of the scope of the current work. We do not cover it in this paper nor in the current implementation of CARDS.

Monitors are the components that carry out the intrusion detection tasks. At the begin-

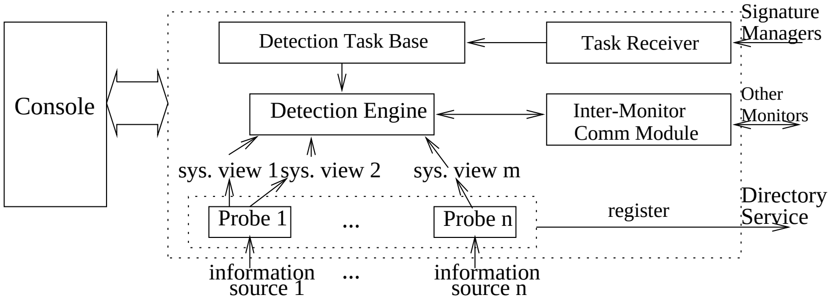


Fig. 14. The monitor architecture

ning of detection, each monitor receives detection tasks from signature managers. During detection, it cooperates with other monitors if some detection tasks are parts of some distributed attacks.

Figure 14 shows the inner structure of a monitor, which is composed of one or more *probes*, a *detection engine*, an *inter-monitor communication module*, a *detection task base*, and a *task receiver*.

Probes are responsible for collecting information from the target system, filtering and reformatting the information into structures defined by system views, and providing the results to the detection engine. Each probe gets information from one particular source, such as a host audit trail. A probe could also be used to derive information from a signature; however, this functionality has not yet been implemented in the current system. The system view configuration of each probe (i.e., system view instances that each probe provides) should be registered in the directory service, which is later retrieved by signature managers to generate specific signatures.

The task receiver receives commands for adding or removing detection tasks from the signature manager, and then add or remove detection tasks from or to the detection task base accordingly.

The detection engine is the core module of the monitor that *executes* the detection tasks. When a detection task is derived from a specific signature involving several monitors, the detection engine cooperates with the detection engines in the related monitors by passing messages through the inter-monitor communication module. The console is the user interface of the monitor.

The directory service is the information center for providing system-wide information to both signature managers and monitors. Two types of information are provided: system view definition and system view configuration. The system view definition specifies the structures and the semantics of the system views. Once a system view is defined, its definition should be placed in the directory service. The system view configuration information specify the system view instances provided by the probes (of the monitors).

The directory service is critical for the scalability of the IDS. It allows the signature managers and the monitors to work in a decentralized and scalable manner and deal with only the components necessary for conducting the designated detection tasks. However, the unavailability of the directory service does not affect the cooperation of detection tasks; instead, it only prevents signature managers from generating specific signatures. Nevertheless, if possible, the directory service should be replicated and distributed (using, for example, LDAP replication server) to provide better availability.

The current version of CARDS is mostly written in Java, with a few probes written in C++ and incorporated into the system via Java Native Interface (JNI). CARDS uses XML to describe system views, signatures, and detection tasks. However, to save the processing time, events transmitted between detection tasks are described using attribute name-value pairs. Since secure communication between the components is not the focus of this system, message transmission between components is carried out over TCP.

We have conducted experiments in small-scale systems. The results showed the feasibility of signature decomposition and the distribution and execution of detection tasks. Further results in large distributed systems are needed to evaluate the scalability of the proposed approaches. In addition, our experience shows that in a large distributed system, it is necessary to have some mechanisms to support various policies regarding the generation of specific signatures from generic ones. However, we do not cover the policy issue in this paper but consider it as future work.

8. CONCLUSION AND FUTURE WORK

In this paper, we explored the abstraction-based misuse detection in distributed environments. We extended the misuse detection model in ARMD [32], which was developed to address the portability of misuse signatures for host-based IDSs, to provide a hierarchical model for distributed attack specification and event abstraction. In addition, we developed a decentralized approach to detect attacks distributed over multiple systems. We also implemented an experimental system called CARDS [58] to examine the feasibility of the proposed approach.

The support provided by our model enables the event abstraction to be dynamic and also benefits the attack specification process. As a misuse detection method, our approach allows signatures to accommodate unknown variants of known attacks. However, this does not imply that our approach can detect entirely new attacks. The unknown attacks that can be detected must share the essential features of some existing signature. In addition, the choice of system views and specification of signatures still depend on the signature writers' understanding of the attacks. Thus, an experienced signature writer may have a good signature that captures the nature of the attack, while a novice may have a narrowly define or even incorrect signature.

The simplicity of the revised model leaded to a decentralized approach to detecting distributed attacks. Considering each event in a signature as a basic processing unit (i.e., detection task), this approach does not require all the information be sent to a central place. Instead, one component IDS needs send messages to another one only when necessary. This greatly reduces the network bandwidth and processing time required to detect distributed attacks. In addition, taking into consideration rare events further improves the detection performance. The same method may be applicable to some anomaly detection models if we can clearly identify the require relation in them. Nevertheless, this requires further work and is out of the scope of this paper.

In addition to the difficulty of writing good signatures, the approach in this paper also has several limitations. Our approach requires reasonably well synchronized clocks in various component systems, continuous operation of each participating component IDS as well as the availability and authenticity of the communication channels between them. If these requirements are not satisfied, systems using our approach may produce false alarms and miss certain attacks. Moreover, a trade-off has to be made about how much historical audit data to keep in each component IDS due to memory constraint and performance reasons. On the one hand, if we decide to keep too much data, a component IDS may not have enough memory to keep the historical data and the performance of a component IDS may degrade. On the other hand, if we do not keep enough historical data, we may miss some stealthy attacks. Nevertheless, these problems are not unique to our approach, but common to all distributed intrusion detection techniques.

The experimental system CARDS provides a test-bed for the abstraction-based approach proposed in this paper. In CARDS, specification of distributed attacks are separated from the detection of the attacks. Distributed attacks are described by generic signatures, which are common to all systems that provide the system views underlying the signatures. To protect specific systems, generic signatures are first mapped to specific signatures using the system configuration information, and then specific signatures are decomposed into detection tasks, which are distributed to and executed by the cooperative component IDSs. Our experience with CARDS showed the feasibility of the abstraction-based approach, but also pointed out more research issues that need to be addressed.

Several issues are worth future research. In this paper, we have studied how to generate and distribute detection tasks from signatures, and the whole process is done in a predefined way. An interesting way that could possibly improve the performance is to dynamically and adaptively generate and distribute detection tasks according to the current detection result. The second issue is to further improve the performance of individual detection tasks. Our current research on distributed detection focuses on the coordination of detection tasks; however, it is equally important to efficiently execute detection tasks. The third issue is the generation of specific signatures from generic ones. Here we generate all possible specific signatures from one generic one according to the current system configuration. A policy language that can control the specific signatures generation process could provide more flexibility from the administration's perspective. Finally, we will continue to refine the implementation of CARDS.

ACKNOWLEDGEMENT

The authors are grateful to the anonymous reviewers for their valuable comments. The authors would also like to thank Dr. Gene Spafford for his suggestions on improving the quality of our paper.

REFERENCES

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, 1980.

[3] R.G. Bace. *Intrusion Detection*. Macmillan Technology Publishing, 2000.

[4] D. Barbará, N. Wu, and S. Jajodia. Detecting novel network intrusion using bayes estimators. In *Proceedings of the First SIAM Conference on Data Mining*, April 2001.

[5] M. Bishop. A security analysis of the NTP protocol version 2. In *Proceedings of the 6th Annual Computer Security Applications Conference*, pages 20–29, 1990.

[6] H.Y. Chang, R. Narayan, C. Sargor, F. Jou, S.F. Wu, B.M. Vetter, F. Gong, X. Wang, M. Brown, and J.J. Yuill. Deciduous: Decentralized source identification for network-based intrusions. In *6th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE Communication Society, 1999.

[7] H.Y. Chang, S.F. Wu, C. Sargor, and X. Wu. Towards tracing hidden attackers on untrusted IP networks. Submitted for publication., 2000.

[8] D. Curry and H. Debar. Intrusion detection message exchange format data model and extensible markup language (xml) document type definition. Internet Draft, draft-ietf-idwg-idmef-xml-03.txt, February 2001.

[9] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Record*, 14(2):1–8, 1984.

[10] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. A common intrusion specification language. http://www.gidos.org/drafts/language.txt, 2000.

[11] R. Feiertag, S. Rho, L. Benzinger, S. Wu, T. Redmond, C. Zhang, K. Levitt, D. Peticolas, M. Heckman, S. Staniford, and J. McAlerney. Intrusion detection inter-component adaptive negotiation. *Computer Networks*, 34:605–621, 2000.

[12] B.S. Feinstein, G.A. Matthews, and J.C.C. White. The intrusion detection exchange protocol (IDXP). Internet Draft draft-ietf-idwg-beep-idxp-02.txt, March 2001.

[13] C. Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54:199–227, 1992.

[14] D. Frincke, D. Tobin, J. McConnell, J. Marconi, and D. Polla. A framework for cooperative intrusion detection. In *Proceedings of the 21st National Information Systems Security Conference*, Crystal City, Virginia, October 1998.

[15] L. T. Heberlein, B. Mukherjee, and K. N. Levitt. Internetwork security monitor: An intrusion-detection system for large-scale networks. In *Proceedings of 15th National Computer Security Conference*, pages 262–271, Baltimore, MD, October 1992.

[16] Y. Ho, D. Frincke, and D. Jr. Tobin. Planning, petri nets, and intrusion detection. In *Proceedings of the 21st National Information Systems Security Conference*, Crystal City, Virginia, October 1998.

[17] J. Hochberg, K. Jackson, C. Stallings, J. F. McClary, D. DuBois, and J. Ford. NADIR: An automated system for detecting network intrusion and misuse. *Computers & Security*, 12(3):235–248, May 1993.

[18] K. Ilgun. USTAT: A real-time intrusion detection system for UNIX. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 16–28, Oakland, CA, May 1993.

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

[20] H.S. Javits and A. Valdes. The NIDES statistical component: Description and justification. Technical report, SRI International, Computer Science Laboratory, 1993.

[21] Y.F. Jou, F. Gong, C. Sargor, X. Wu, S.F. Wu, H.C. Chang, and F. Wang. Design and implementation of a scalable intrusion detection system for the protection of network infrastructure. In *DARPA Information Survivability Conference and Exposition*, 2000.

[22] C. Kahn, D. Bolinger, and D. Schnackenberg. Communication in the common intrusion detection framework. http://www.gidos.org/drafts/communication.txt, 1998.

[23] C. Kahn, P. A. Porras, S. Staniford-Chen, and B. Tung. A common intrusion detection framework. Submitted to Journal of Computer Security, July 1998.

[24] R. A. Kemmerer. NSTAT: A model-based real-time network intrusion detection system. Technical Report TRCS97-18, Reliable Software Group, Department of Computer Science, University of California at Santa Barbara, 1997.

[25] K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, Department of EECS, MIT, June 1999.

[26] F. Kerschbaum, E.H. Spafford, and D. Zamboni. Using embedded sensors for detecting network attacks. In *Proceedings of the 1st ACM Workshop on Intrusion Detection Systems*, November 2000.

[27] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, August 1995.

[28] 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, October 1994.

[29] W. Lee, R.A. Nimbalkar, K.K. Yee, S.B. Patil, P.H. Desai, Tran T.T., and S.J. Stolfo. A data mining and CIDF based approach for detecting novel and distributed intrusions. In *Proceedings of 3rd International Workshop on the Recent Advances in Intrusion Detection*, October 2000.

[30] W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *Proceedings 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999. To appear.

[31] J. Lin. *Abstraction-Based Misuse Detection: High-level Specifications and Adaptable Stragegies*. PhD thesis, George Mason University, Fairfax, VA, December 1998.

[32] J. Lin, X. S. Wang, and S. Jajodia. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the 11th Computer Security Foundations Workshop*, pages 190–201, Rockport, MA, June 1998.

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

[34] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, University of Namur (Belgium), September 1997.

[35] A. Mounji, B.L. Charlier, D. Zampuniéris, and N. Habra. Distibuted audit trail analysis. In *Proceedings of the ISOC '95 Symposium on Network and Distributed System Security*, pages 102–112, 1995.

[36] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May 1994.

[37] V. Neufeldt, editor. *Webster's New World Dictionary of Amercian English*. Webster's New World, 3rd college edition edition, 1988.

[38] D. New. The TUNNEL profile. Internet Draft draft-ietf-idwg-beep-tunnel-01.txt, February 2001.

[39] P. Ning, X. S. Wang, and S. Jajodia. Modeling requests among cooperating intrusion detection systems. *Computer Communications*, 23(17):1702–1716, 2000.

[40] P. Ning, X. S. Wang, and S. Jajodia. A query facility for common intrusion detection framework. In *Proceedings of 23rd National Information Systems Security Conference*, pages 317–328, Baltimore, MD, 2000.

[41] S. Northcutt. *Network Intrusion Detection: An Analyst's Handbook*. New Riders, 1999.

[42] P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. The common intrusion detection framework architecture. http://www.gidos.org/drafts/architecture.txt, 1998.

[43] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling response to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, National Institute of Standards and Technology, 1997.

[44] M. Rose. The blocks extensible exchange protocol core. IETF RFC 3080., March 2001.

[45] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *Proceeding of 1997 IEEE Symposium on Security and Privacy*, pages 208–223, Oakland, CA, May 1997.

[46] S. E. Smaha. Haystack: An intrusion detection system. In *Proceedings of Fourth Aerospace Computer Security Applications Conference*, December 1988.

[47] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (distributed intrusion detection system) - motivation, architecture, and an early prototype. In *Proceedings of 14th National Computer Security Conference*, pages 167–176, Washington, D.C., October 1991.

[48] TimesTen Performance Software. Architecture for real-time data management: Timesten's core in-memory database technology. White Paper, 2001.

[49] E.H. Spafford and D. Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34:547–570, 2000.

[50] 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 *Proceedings of the 19th National Information Systems Security Conference*, volume 1, pages 361–370, October 1996.

[51] S. Staniford-Chen and L. Heberlein. Holding intruders accountable on the internet. In *Proceedings of 1995 IEEE Symposium on Security and Privacy*, pages 39–49, Oakland, May 1995.

[52] Smith. S.W. and J.D. Tygar. Security and privacy for partial order time. In *ISCA Seventh International Conference on Parallel and Distributed Computing Systems*, October 1994.

[53] J. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, April 1997.

[54] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

[55] G. Vigna and R. A. Kermmerer. NetSTAT: A network-based intrusion detection approach. In *Proceedings of the 14th Annual Security Applications Conference*, December 1998.

[56] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network*, pages 20–23, January 1996.

[57] S.F. Wu, H.C. Chang, F. Jou, F. Wang, F. Gong, C. Sargor, D. Qu, and R. Cleaveland. JiNao: Design and implementation of a scalable intrusion detection system for the OSPF routing protocol. To appear in Journal of Computer Networks and ISDN Systems.

[58] J. Yang, P. Ning, X. S. Wang, and S. Jajodia. CARDS: A distributed system for detecting coordinated attacks. In Sihan Qing and J. H. P. Elof, editors, *Proceedings of IFIP TC11 Sixteenth Annual Working Conference on Information Security (SEC 2000)*, pages 171–180. Kluwer Academic Publishers, August 2000.

APPENDIX

A. PROOF OF THEOREM 1

PROOF. Consider a set of detection tasks derived from a signature *Sig* = (*N*, *E*, *SysView*, *Label*, *Assignment*, *TimedCondition*, *PositiveNodes*) and a workflow tree *T* for *Sig*.

Suppose there is a match of *Sig*. This implies that for each positive node n in *Sig* there is an event, denoted e_n, on the system view associated with n such that they together satisfy the three conditions specified in definition 5. Consider the detection task DT_l for a leaf node n_l in *T*. The $cond$ component of DT_l is the timed condition associated with n_l according to definition 11. Since the event e_{n_l} satisfies the timed condition associated with n_l (which equals to $cond$) and there is no child detection task, the PMT table in step 4 of the algorithm will contain the values of the variables required by other detection tasks. Since the communication is resilient, the variable values in PMT will be sent to the detection task for the parent node of n_l.

Now consider the detection task DT_i for an interior node n_i. Event e_{n_i} will be saved in the history table h at step 6 of the algorithm *DetectionTask*. If DT_i receives the variable values assigned from e_{n_c} for all child nodes n_c of n_i, it will compute a non-empty PMT table in either step 4 (when it receives e_{n_i} after all the variable values) or step 8 (other situations), since e_{n_i} and e_{n_c}'s are part of the match. The table PMT will contain the variable values assigned from e_{n_i} (through the renaming operation) or inherited from the detection tasks for n_i's child nodes. Then in step 14, DT_i will send the variable values to its parent detection task. By induction, the detection task for the positive root node in *T* will store the timestamps of e_n's. This is to say that all matches of the signature will be detected by the algorithm *DetectionTask*.

For each tuple t in the matched table m, t has timestamp attributes $begin\_time_n$ and end_time_n for each positive node n in *Sig*, which together identify an event e_n on the system associated with n. For each node n, since $begin\_time_n$ and end_time_n are finally transmitted to the detection task for the positive root, e_n must satisfy the $cond$ component of the corresponding detection task. According to definition 11, the $cond$ component of each detection task is the conjunction of the timed condition associated with n and the qualitative temporal relationships between n and n's descendants in *T*. This has two implications. First, e_n satisfies the timed condition associated with n. Second, for each arc (n_1, n_2) in *Sig*, either n_1 (when n_1 is closer to the root) or n_2 (when n_2 is closer to the root) satisfying the $cond$ component implies that e_{n_1} and e_{n_2} satisfy the qualitative temporal relationship represented by the labeled arc. This means that there is a match of all the positive nodes.

If $N \neq PositiveNodes$ and there exists a match of all the nodes in *Sig*, the detection task for the negative root in *T* will discover this match and sends the timestamp variables to the positive root. Then this match will be removed from m in step 11. This is to say that all the tuples that remain in m long enough (i.e., longer than the time required to process and transmit the negative events) represent matches of the signature *Sig*. This concludes the proof. □