Design and Implementation of A Decentralized Prototype System for Detecting Distributed Attacks

Peng Ning [a,*] Sushil Jajodia [b] Xiaoyang Sean Wang [b]

[a] *Department of Computer Science*
North Carolina State University, Raleigh, NC 27695

[b] *Center for Secure Information Systems*
George Mason University, Fairfax, VA 22030

Abstract

This paper presents the design and implementation of a decentralized research prototype intrusion detection system (IDS) named CARDS (Coordinated Attacks Response & Detection System), which aims at detecting distributed attacks that cannot be detected using data collected at any single place. CARDS adopts a signature-based approach. It consists of three kinds of independent but cooperative components: *signature manager, monitor,* and *directory service*. Unlike traditional distributed IDSs, CARDS decomposes global representations of distributed attacks into smaller units (called *detection tasks*) that correspond to the distributed events indicating the attacks, and then executes and coordinates the detection tasks in the places where the corresponding events are observed.

Key words: decentralized intrusion detection, misuse detection, computer security, network security

1 Introduction

With rapidly growing connectivity of the Internet, networked computer systems are playing increasingly vital roles in our modern society. While the Internet has brought great benefits to this society, it has also made vital systems vulnerable to malicious attacks [16]. Distributed and coordinated attacks

* Corresponding author. Tel. +1 (919)513-4457
Email addresses: ning@csc.ncsu.edu (Peng Ning), jajodia@gmu.edu (Sushil Jajodia), xywang@gmu.edu (Xiaoyang Sean Wang).

(e.g., the Mitnick attack [35]) are increasingly popular among hackers; such attacks are difficult to detect and to defend against.

The conventional approach to securing a computer or network system is to build "protective shields" around it (e.g., firewall and Virtual Private Network (VPN)). Outsiders who need to enter the system must be identified and authenticated. Also, the "shields" should prevent leakage of information from the protected domain to the outside world [30]. However, such preventive approaches are usually not sufficient to protect computer and network systems due to (potential) design and implementation flaws; thus, intrusion detection was introduced as a second line of defense along with the "protective shields" [10,30].

Many intrusion detection systems (IDSs) have been developed to perform intrusion detection in distributed or networked systems (e.g., ASAX [29], EMERALD [36], GrIDS [42]). The early systems (e.g., ASAX [29] and NSTAT [22]) require the audit data collected from different places be sent to a central location for analysis. Although the audit data is usually reduced before being transmitted, the scalability of such systems is still limited due to the centralized analysis. In order to improve the scalability, later systems (e.g., EMERALD [36], AAFID [40]) propose to deploy multiple component IDSs in different locations and organize them into a fixed hierarchy, where the low-level IDSs send designated information to high-level IDSs. The hierarchical approach certainly scales better than the previous centralized method; 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 finally reach a common high-level IDS and be correlated together. 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 the design and implementation of a research prototype IDS named CARDS (Coordinated Attack Response & Detection System). CARDS was developed to examine the decentralized intrusion detection technique, which was proposed as a part of the abstraction-based intrusion detection in [32]. CARDS aims at detecting the distributed attacks that cannot be detected using data collected from any single location. Unlike previous distributed IDSs, CARDS decomposes global representations of distributed attacks into smaller units (called *detection tasks*) that correspond to the distributed events indicating the attacks, and then executes and coordinates the detection tasks in the places where the corresponding events are observed. In particular, the message transmission between component IDSs is not determined by a centralized or hierarchical scheme; instead, one component IDS

sends a message to another only when the message is required by the later IDS to detect certain attacks. Thus, the communication cost imposed by intrusion detection is greatly reduced in CARDS.

The rest of the paper is organized as follows. The next section presents the related work. Section 3 briefly describes the abstraction-based intrusion detection, which is the foundation of CARDS. Section 4 presents the architecture of CARDS. Section 5 describes the approaches that CARDS uses to perform decentralized intrusion detection. Section 6 presents the implementation of the current version of CARDS. Section 7 concludes the paper and points out the future research plan.

2 Related work

Intrusion detection has been studied for about twenty years since the Anderson's report [2]. A survey of the early work on intrusion detection is given in [30], and an excellent overview of the current intrusion detection techniques and related issues can be found in a recent book [3]. Among all these techniques, our work is closely related to those developed for distributed intrusion detection.

Early distributed intrusion detection systems collect audit data from distributed component systems but analyze them in a central place (e.g., DIDS [38], NADIR [17], NSTAT [22] and ASAX [29]). 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.

Several systems have been developed recently to address the scalability issue (e.g., EMERALD [36], GrIDS [42], and AAFID [40]). EMERALD uses both misuse detection and statistical anomaly detection techniques and adopts a recursive framework in which generic building blocks can be deployed in a highly distributed manner [36]. GrIDS performs intrusion detection by aggregating computer and network information into activity graphs which reveal the causal structure of network activity [42]. As a generic distributed intrusion detection platform, AAFID consists of four types of components: agents, filters, transceivers and monitors [40], which are organized in a tree structure with child and parent components communicating with each other. CARDS differs from these systems as follows: While CARDS decomposes and coordinates distributed event collection and analysis according to the relationships between the distributed events, the aforementioned systems rely on some predefined hierarchical organization, which is usually determined by administrative concerns (e.g., EMERALD [36], GrIDS [42]). Compared with the hierarchical approach, our approach has the advantage that the component IDSs can ex-

change necessary information without forwarding it along the hierarchy.

NetSTAT is an application of STAT [18] to network-based intrusion detection [44,45]. 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. CARDS is similar to NetSTAT in that both approaches can decide what information needs to be collected in various places. However, CARDS 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, CARDS analyzes the distributed events in a decentralized way, that is, the events are analyzed as being collected in various places.

JiNao is an IDS that detects intrusions against network routing protocols [20,46]. 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 [20,46]; however, no specific mechanisms have been provided for doing so. In contrast, CARDS is a generic decentralized IDS and provides a specific way to coordinate intrusion detection activities in different places.

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 [21]. Several efforts have tried to further 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 [12]; 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 [24]; finally, the query facility for CIDF enables CIDF components to request specific information from each other [33,34].

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. The current results include an Intrusion Detection Message Exchange Format (IDMEF) [8], an Intrusion Detection Exchange Protocol (IDXP) [13], and a Tunnel profile for different systems to exchange messages through firewalls [31].

We view CIDF, IDMEF (IDXP) and their extensions as complementary to ours. Though providing common message formats and exchange procedures, neither CIDF nor IDWG has 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 [24]. 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 [15]. 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. 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, CARDS 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.

CARDS is a test-bed used to examine the decentralized, abstraction-based intrusion detection proposed in [32], which was extended from a host-based misuse detection system named ARMD [26,27]. In [32], we have examined the issues related to representation of distributed attacks and decentralized detection of distributed attacks. In this paper, we focus on the design and implementation issues as well as our experiences with CARDS.

There are many other related work, such as various anomaly detection models (e.g., NIDES/STAT [19], HAYSTACK [37]), data mining approaches (e.g., JAM [25], ADAM [4]), various tracing techniques (e.g., DECIDUOUS [7], thumbprinting [43]), and embedded sensors [23]. We consider these techniques as complementary to ours presented in this paper.

3 Abstraction-based Intrusion Detection

CARDS is a research prototype system developed to examine the research issues involved in the abstraction-based intrusion detection model, which is proposed in [32]. This section gives a brief overview of this model.

There are three essential concepts in the abstraction-based intrusion detection 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 an optional set of

dynamic predicates. The event schema specifies the attributes that describe the (abstract) events on a system view, while the dynamic predicates tell the relationship among system entities at certain times. For example, a network monitor that reports Denial of Service (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*}. 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. 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 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.

An *event* on a system view is a tuple on the event schema with an interval timestamp [*begin_time*, *end_time*], where the tuple consists of the event attribute values and the timestamp represents the interval when the event occurs. For example, we may have an event e on *TCPDOSAttacks* whose attributes are: *VictimIP = 129.174.142.177, VictimPort = 80, begin_time = 1, end_time = 20.* A finite set of events on a system view and the Boolean functions that evaluate the dynamic predicates for the time when these events occur are together called an *event history* on the system view.

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.

To simplify the specification of signatures, we use the qualitative temporal relationships between events. A qualitative temporal relationship is a well-defined relation between two events. Examples include simple relationships such as `before`, `after`, `during` as well as their logical combinations (e.g., `before` OR `during`). Figure 1 gives a list of simple qualitative temporal relationships and their definitions. (For details please refer to [1,14,32].) 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.

We model a signature as a labeled directed graph. Each node in the graph corresponds to an event on a particular system view instance and each arc is labeled with 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*

relation	*meaning*	*inverse relation*
e_1 `equal` e_2	$e_1.begin\_time = e_2.begin\_time$	`equal`
	and $e_1.end\_time = e_2.end\_time$	
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	$e_1.begin\_time < e_2.begin\_time$	`inv-overlaps`
	and $e_1.end\_time < e_2.end\_time$	
	and $e_1.end\_time > e_2.begin\_time$	
e_1 `during` e_2	$e_1.begin\_time > e_2.begin\_time$	`inv-during`
	and $e_1.end\_time < e_2.end\_time$	
e_1 `starts` e_2	$e_1.begin\_time = e_2.begin\_time$	`inv-starts`
	and $e_1.end\_time < e_2.end\_time$	
e_1 `finishes` e_2	$e_1.begin\_time > e_2.begin\_time$	`inv-finishes`
	and $e_1.end\_time = e_2.end\_time$	
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	$e_1.begin\_time < e_2.end\_time$	`contemporary`
	and $e_2.begin\_time < e_1.end\_time$	
e_1 `born-before-death` e_2	$e_1.begin\_time < e_2.end\_time$	`die-after-birth`

Fig. 1. The simple qualitative temporal relationships between two events

with each node. We also associate with each node a set of assignments of event attributes to variables in order for the timed conditions of other nodes to refer to event attribute values of this node. As a restriction, each variable can only appear in one assignment in a signature. Furthermore, we distinguish two kinds of nodes: *positive nodes* and *negative nodes*. Positive nodes represent events that are necessary for an attack (which we call *positive events*), while negative nodes represent such events (which we call *negative events*) that if they coexist with the positive events, then the positive events do not constitute an attack.

Figure 2 shows a generic version of the signature for the Mitnick attack de-

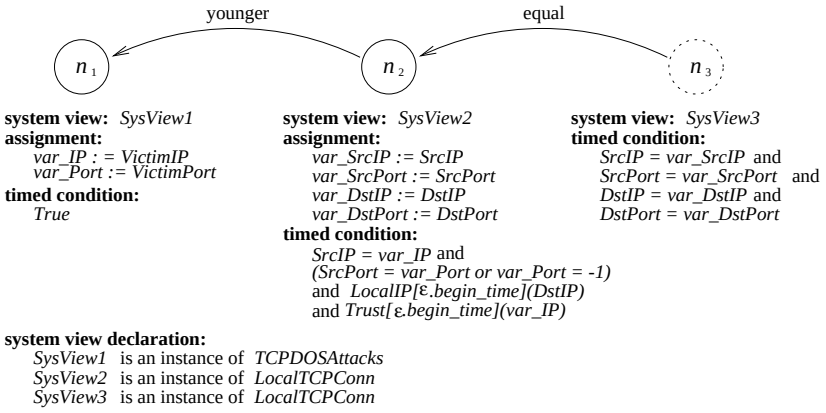


Fig. 2. The signature for the Mitnick attack

scribed in [35]. In such an attack, the attacker first launches a DOS attack to prevent a trusted host from accepting incoming TCP connection requests (i.e., SYN packets), and then tries to connect to another (trusting) host using the IP address and TCP port being flooded as source IP and source port (via IP spoofing). The signature involves three system view instances: an instance of the system view *TCPDOSAttacks* and two instances of *LocalTCPConn*, which represent the observable information about the DOS attack, the TCP connections on the trusting host and the TCP connections on the trusted host, respectively. Node n_1 and n_2, which are depicted with solid circles, 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 the trusting host. 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 the trusting host, and the source host is trusted. The labeled arc (n_2, n_1) restricts that this TCP connection should occur after the begin time of the DOS attack. Node n_3, which is drawn in dotted circle, represents a negative event and stands for a TCP connection observed on the trusted host being DOS attacked. 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 observe the same connection.

A signature in our model specifies a generic pattern of a certain type of attacks, which is usually independent of any specific systems. 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 particular system 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 2 is a generic signature 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, it becomes a specific signature representing the Mitnick attack on these systems.

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.

A view definition is used to derive information from the matches of a signature and present it through a system view. For example, suppose we modify the signature *Mitnick* in figure 2 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 system view derived by the above view definition is ({*Attack, VictimHost, VictimPort, TrustedHost*}, {*Trust[t](var_host)*}).

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 *TCPDOSAttacks* 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 the abstraction-based is how to detect the specified distributed attacks. In [32], we proposed an approach to organizing autonomous but cooperative component systems to detect dis-

tributed attacks in a decentralized way. Specifically, we decompose global representations of distributed attacks into smaller units called *detection tasks*, each of which corresponds to one type of distributed events possibly involved in the attack and is executed in the place where the events are observed. The detection tasks are then coordinated to achieve the same result as they were performed in a central location. For example, the signature shown in figure 2 can be decomposed into three detection tasks: n_1, n_2, and n_3, which will be executed in the systems where the corresponding events are observed. CARDS is the research prototype that we developed to examine the feasibility of this approach and further identify possible research issues.

4 CARDS Architecture

CARDS is a distributed intrusion detection system composed of three kinds of independent but cooperative components: *signature manager*, *monitor* and *directory service*. Figure 3 shows the architecture of CARDS. In a typical environment, there may be one or more signature managers and monitors. The monitors can be embedded in the monitored system or as a dedicated monitor running on a separate system from the monitored system. Several monitors can cooperate with each other through message passing when they are involved in detecting one specific signature. A typical configuration of CARDS is depicted in Figure 4. The details of these components are presented in the following subsections.

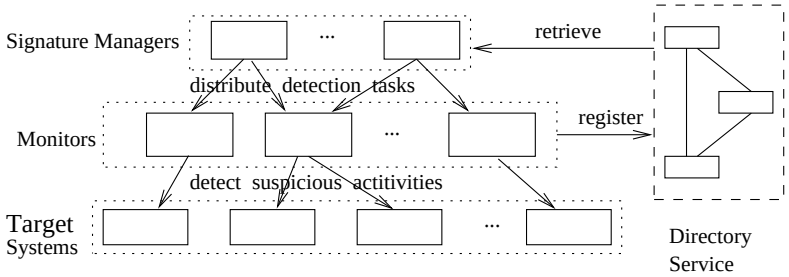


Fig. 3. The CARDS architecture

4.1 Signature Manager

Signature managers are the administrative components in CARDS. As shown in figure 3, with the monitor configuration information retrieved from the directory service, a signature manager (1) generates specific signatures from generic signatures, (2) decomposes specific signatures into intrusion detection tasks, and (3) distributes these tasks to the involved monitors.

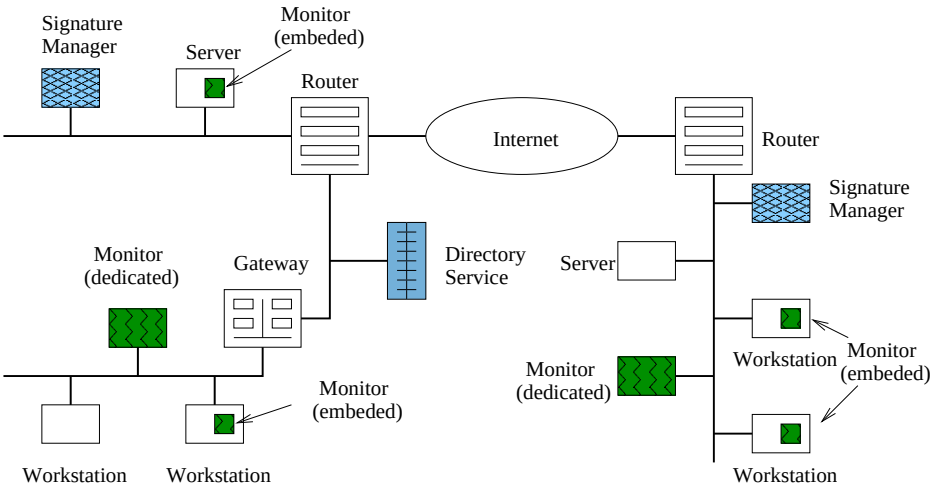


Fig. 4. A typical configuration

The technical challenges involved in signature managers is how to generate specific signatures from generic one and how to decompose specific signatures into detection tasks. We will discuss specific signatures generation and decomposition in section 5.

The distribution of detection tasks can be carried out on the basis of existing transport services such as TCP. (The current implementation of CARDS is based TCP protocol using Socket.) In a real deployment, the communication between signature managers and monitors should be secured. In particular, the source and the content of each detection task should be authenticated. Encryption of detection tasks is not essential; however, in some extremely secure applications where it is not desirable for the enemy to know what the IDSs are doing, it might be necessary encrypt the detection tasks as well.

4.2 Monitor

The monitors are the components that carry out the intrusion detection tasks. In practice, each monitor is an application that can run on the monitored system or a separate system apart from the monitored system. Each monitor does audit information collection, filtering, reformatting as well as detecting the attacks. A monitor receives the detection task from a signature manager. During the process of executing a detecting attack, it may cooperate with other monitors as specified by the detection task.

Figure 5 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*, a *task receiver*, and a *console*.

Probes are responsible for collecting information from the target system, fil-

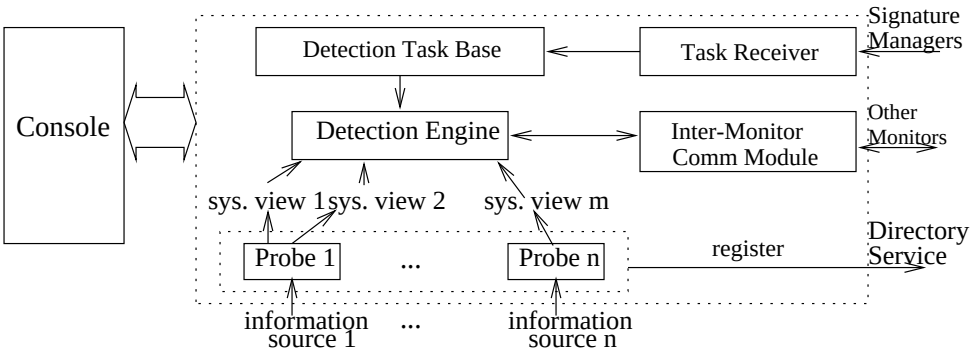


Fig. 5. The monitor architecture

tering and reformatting the information according to the system view as well as providing the results to the detection engine. Each probe gets information from one particular information source such as a host audit trail and the traffic on a network segment. While one monitor may have one or more probes, each probe may also provide one or more system views, each of which corresponds to one particular aspect of the information source. For example, a probe that gets information from the host audit trail may have two system views: one for the user's shell commands, and the other for the TCP connections that involve the host. The system view configuration of each probe (i.e., what system views 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. We will discuss how the detection engines works in section 5.4. The console is the user interface of the monitor.

4.3 Directory Service

The directory service is the central place for providing system-wide information to both signature managers and monitors. Although the directory service may be distributed or replicated, they function as a single component of CARDS. Thus, the signature managers and the monitors are allowed to work in a decentralized and scalable manner and deal with only the components necessary for conducting the designated detection tasks.

The directory service provides two types of information: *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 specifies the system view instances provided by the probes (of the monitors). As described earlier, the probe registration module of the monitors updates this information when a monitor is deployed or reconfigured. The signature manager needs to refer to this information when it generates specific signatures.

The purpose of the directory service is to make our system scalable. Since distributed attacks usually consist of several sessions that spread across several systems and cannot be reliably detected from a single place, the monitors that detect them should be installed at various places, possibly across several different administrative domains. To achieve the scalability, interoperability, and distribution of the system, we adopt the directory service to provide global information that needs to be shared by the signature managers and the monitors. As a result, the signature managers and the monitors are allowed to work in a decentralized way and deal with only the components that are necessary to perform the designated detection tasks.

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 monitors; 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.

5 Cooperative Detection of Distributed Attacks

In this section, we discuss the approaches that signature managers use to generate and decompose specific signatures and the procedures with which monitors cooperatively detect the coordinated attacks. Theoretical aspect of this work has been discussed in [32]. Here we focus on system aspects and discuss the design and implementation issues in the framework of CARDS.

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE system_view SYSTEM "http://infosec/sysview.dtd">
<system_view name="TCPDOSAttacks">
    <event_schema>
        <attribute name="VictimIP" type="varchar" len="15"/>
        <attribute name="VictimPort" type="int" len="1"/>
    </event_schema>
</system_view>
```

Fig. 6. The system view *TCPDOSAttacks*

5.1 Internal Languages

Since CARDS is a distributed IDS consisting of multiple components that need to communicate with each other, it is necessary to enable different components to understand each other. In other words, CARDS components need some common languages. Due to the rich semantics typically involved in the cooperation of signature managers and monitors, simple agreements are usually not good enough to provide the necessary support. In addition, for the convenience of configuration and ease of experiments, it is desirable that the messages between CARDS components are not only computer comprehensible, but also human readable.

As discussed in section 2, several efforts are under way to establish some common languages (e.g., IDMEF [8]) for IDSs. However, all the existing results do not cover the ground required by CARDS components. This is because all the existing results do not consider any specific mechanisms to decompose, distribute, and coordinate decentralized intrusion detection activities, which are central issues studied in CARDS. Thus, we decided to develop a set of internal languages for the CARDS components.

Our internal languages are based on eXtensible Markup Language (XML) [5]. We chose XML as the foundation of the internal languages due to the following reasons: (1) XML is not only computer comprehensible, but also human readable; (2) XML is widely accepted in many areas (e.g., IDMEF [8]) and could help in possible future technology transfer; (3) There are rich API supports for XML related development (e.g., Apache's Xerces parsers).

We defined Document Type Definitions (DTDs) for system views, signatures, and detection tasks. Due to space reasons, we cannot include the details of these DTDs in this paper. Figures 6 through 8 show example XML representations (with some details omitted) of system views, signatures and detection tasks, respectively.

The current focus of CARDS is to examine the decentralized detection procedure proposed in [32]; thus, view definition is not supported yet. We plan to

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE signature SYSTEM "http://infosec/signature.dtd">
<signature name="mitnick">
    <node name="n1">
        <system_view_inst name="SysView1"/>
        <assignment>
            <variable name = "var_IP"/> <attribute name="VictimIP"/>
        </assignment>
            ...
    </node>
    <node name="n2">
        <system_view_inst name="SysView2"/>
        <assignment>
            <variable name="var_SrcIP"/> <attribute name="SrcIP"/>
        </assignment>
            ...
        <timed_condition>
            <and>
                <equal>
                    <attribute name="SrcIP"/> <var_ref name="var_IP"/>
                </equal>
                    ...
            </and>
        </timed_condition>
    </node>
    <node name="n3" type="negative">
        <system_view_inst name="SysView3"/>
        <timed_condition> ... </timed_condition>
    </node>
    <edge from="n2" to="n1" label="younger"/>
    <edge from="n3" to="n2" label="equal"/>
    <declare_sys_view_inst name="SysView1"
                           system_view="DOSAttacks"/>
    <declare_sys_view_inst name="SysView2" ... />
    <declare_sys_view_inst name="SysView3" ... />
</signature>
```

Fig. 7. The generic signature for the Mitnick attack (some details omitted)

develop the corresponding DTD for view definition when we need to perform the related experiments. In addition, the DTD for signatures does not support complex qualitative temporal relationships between events (e.g., e_1 (`before` OR `during`) e_2); however, this feature does not affect the decentralized detection very much and can be easily incorporated into the current DTD.

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE task SYSTEM "http://infosec/task.dtd">
<task name="n2" root="yes" spec_sig_name="mitnick_1" type="positive">
    <system_view_inst name="SysView2" system_view="LocalTCPConn">
        <monitor baseport="4200" location="129.174.142.177" name="m2"/>
        <probe name="LocalTCPConn"/>
    </system_view_inst>
    <partial_match_table>
        <PMT_attribute len="1" name="var_DstPort" source="assign"
                source_attr="DstPort" type="int"/>
        ...
    </partial_match_table>
    <timed_condition> ... </timed_condition>
    <parent_node name="n3">
        <monitor baseport="4200" location="129.174.142.140" name="m3"/>
    </parent_node>
    <child_node name="n1">
        <monitor baseport="4200" location="129.174.142.140" name="m1"/>
        <child_partial_match_table>
            <child_PMT_attribute name="begin_time_n1" ... />
            <child_PMT_attribute name="end_time_n1" ... />
            <child_PMT_attribute name="var_IP" ... />
            <child_PMT_attribute name="var_Port" ... />
        </child_partial_match_table>
    </child_node>
    <negative_root_node name="n3">
        <monitor baseport="4200" location="129.174.142.140" name="m3"/>
        <negative_root_partial_match_table>
            <negative_root_PMT_attribute name="end_time_n3" .../>
            ...
        </negative_root_partial_match_table>
    </negative_root_node>
</task>
```

Fig. 8. Detection task n_2 of a specific signature for the Mitnick attack (some details omitted)

5.2 Specific Signature Generation

One of the central tasks of signature managers is to generate specific signatures from a generic one. For the sake of presentation, we first define some terms. We say that a probe *has* the system view v if the probe provides information through v. We say a system view instance v of a signature is *associated* with the probe p of the monitor m if p is designated to provide information for the signature through v. In CARDS, a signature is called a specific signature if each system view instance of the signature is associated with a probe of a monitor in the system. Thus, the task of generating specific signatures from a

Table 1
A list of monitors, probes and their system views

Monitor	*Probe*	*System View*
Sniffer	DOSProbe	TCPDOSAttacks
Megalon	MegalonTCPProbe	LocalTCPConn
Meadow	MeadowTCPProbe	LocalTCPConn
Infosec	InfosecTCPProbe	LocalTCPConn
Backeast	BackeastTCPProbe	LocalTCPConn

generic one is to associate all the system view instances of the signature with the probes of the monitors in the system.

When generating specific signatures from a generic one, the signature manager first looks up in the directory service for the probes that have the system views whose instances are used in the generic signature. Then the signature manager derives specific signatures by associating the system view instances with the probes of the monitors under its control. For example, consider the signature shown in figure 2. Suppose there are a signature manager and three monitors called *Sniffer*, *Megalon* and *Backeast*. If the signature manager learns by looking up the directory service that the three monitors have three probes *DOSProbe*, *MegalonTCPProbe*, and *BackeastTCPProbe* that have the system view *TCPDOSAttacks*, *LocalTCPConn*, and *LocalTCPConn*, respectively, then it can generate a specific signature by associating the *SysView1*, *SysView2*, and *SysView3* with these probes. This specific signature then describes the Mitnick attack against the hosts monitored by *Megalon* and *Backeast*.

One generic signature usually generates more than one specific signature, since the attacks modeled by the generic signature may happen against different targets. In the previous example, we can generate another specific signature by associating *SysView2* with *BackeastTCPProbe* and *SysView3* with *MegalonTCPProbe*, which represents a different attacking strategy against the two hosts. In addition, there may be other hosts being monitored, and the Mitnick attack may be launched against these hosts as well. Thus, the signature manager should associate the system view instances of a generic signature with all combinations of probes where the attack may happen to the targets monitored by the probes.

Consider the generic signature for the Mitnick attack shown in figure 2. Suppose in the directory service the monitors and the probes that support the system views underlying the signature are as shown in Table 1. When generating specific signatures for the Mitnick attack, the signature manager first

retrieves this information from the directory service and then associate the system view instances of the generic signature with the probes in all possible combinations that the Mitnick attack may happen. For this particular probe configuration, the signature manager will generate 12 specific signatures, with *SysView1* associated with the probe *DOSProbe* at the monitor *Sniffer*, and *SysView2* and *SysView3* with all combinations of the rest of the probes.

Some optimization may be achieved using the timed condition. In the above example, the specific signature that associates *SysView2* with *MegalonTCP-Probe* and *SysView3* with *BackeastTCPProbe* needs to be generated only if the host monitored by *Megalon* trusts the host monitored by *Backeast*. Unfortunately, such optimization can only be performed manually in the current version of CARDS.

The reader may have noticed a potential problem with the specific signature generation: There could be too many specific signatures due to the combinatorial nature of the generation procedure. Although we have argued that an attack can potentially happen to each of the specific signatures, having too many specific signatures may introduce severe performance penalty into the system. One method to alleviate the problem is to introduce the notion of *managed domain*, which restricts the number of monitored nodes in each domain. Moreover, it is worth noting that in reality, people care more about protecting critical resources such as gateways and file servers. Thus, it is desirable to have certain ways (e.g., a policy language) with which the system administrator can describe what (combinations of) systems should be protected. Such methods can help reduce the number of specific signatures; however, they are out of the scope of this paper.

5.3 Specific Signature Decomposition

In [32] we have discussed various choices of specific signature decomposition as well as the related issues, and developed an approach to decomposing specific signatures into detection tasks. This approach has been used in the implementation of CARDS. In the following, we first briefly describe this approach and then give some details about its implementation in CARDS.

Our signature decomposition is based on a dependency relation named *require* between the nodes (i.e., events) involved in a signature. Intuitively, 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, n *directly requires* n' if n requires some variables assigned at n'. Moreover, 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 2 n_2 requires n_1, and n_3 requires both n_2

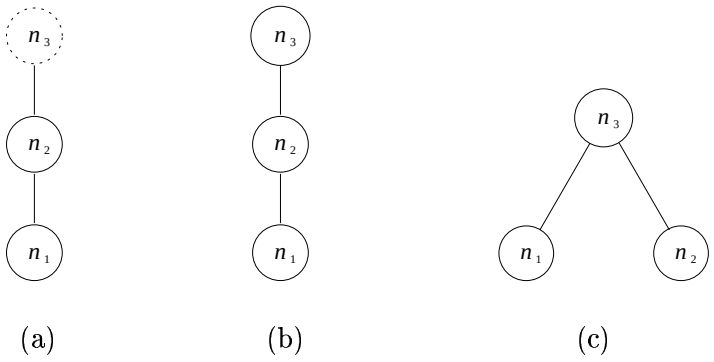


Fig. 9. Examples of workflow trees

and n_1. Intuitively, node n needs information from node n' through variable assignments if n (directly) requires n'.

The decomposable signatures have been identified as the *serializable* signatures [32]. 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. For example, the signature in figure 2 is serializable: The require relation 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. 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.

We use a structure called workflow tree to help the decomposition of a specific signature. 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 2.

In a workflow tree, the root of the subtree that contains all and only 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.

A serializable signature can be decomposed with a given workflow tree. The idea of signature decomposition is to have each node (i.e., event) of the signa-

ture as a basic execution unit, that is, a detection task, and coordinate these detection tasks to detect the distributed attacks represented by the signature.

The first step of signature decomposition is to transform the qualitative temporal relationships represented by the labeled arcs into specific conditions and incorporate them into the timed conditions. This is done using the timestamp variables. For each node n, we add two additional assignments: $begin\_time\_n := begin\_time$ and $end\_time\_n := end\_time$, where $begin\_time$ and end_time are the interval-based timestamp. (Since timestamps contain critical time information of the events, we perform the assignments even if no labeled arc involves the node (event).) If there is a labeled arc from n_1 to n_2, for example, n_1 `before` n_2, the labeled arc can be transformed into $end\_time\_n_1 < begin\_time\_n_2$. Then we modify the timed condition associated with the least common ancestor [1] of n_1 and n_2 as the conjunction of the previous timed condition and $end\_time\_n_1 < begin\_time\_n_2$.

Since all the nodes in a signature are related to each other via variable assignments, a critical job in decomposing a signature is to identify what information of a detection task is required by other detection tasks through variables. This can be done by a depth-first traversal of the given workflow tree. To do this, we associate with each node n two sets: $RequiredSet_n$ which contains variables required by n's ancestors, and $ProvidedSet_n$ which contains variables assigned at n and n's descendants. It is easy to see that the variables that a node n needs to send to its parent is $ProvidedSet_n \cap RequiredSet_n$.

To assist in the computation of $ProvidedSet_n$ and $RequiredSet_n$ for each node n, we associate with n two additional sets: $AssignedSet_n$ which contains the set of variables assigned at n, and $NeededSet_n$ which contains the set of variables used in the timed condition of n. Both $AssignedSet_n$ and $NeededSet_n$ can be easily computed for each node n.

Initially, $RequiredSet_{root} = \emptyset$, and $ProvidedSet_l = AssignedSet_l$, where l is a leaf node. When traversing down the workflow tree, each node n passes $RequiredSet_n$ to its chid nodes along with the variables directly used in its timed condition. In other words, the child node n' of n will have $RequiredSet_{n'} = RequiredSet_n \cup NeededSet_n$. When traversing up the workflow tree, each node n gets $ProvidedSet_{n'}$ for all child nodes n', and thus $ProvidedSet_n = \cup_{\forall n' s\, child\, n'} ProvidedSet_{n'} \cup AssignedSet_n$. It then returns $ProvidedSet_n$ to its parent.

Once the variables that each node needs to send to its parent node are identified, the corresponding signature can be easily decomposed. In order to cooperate with other detection tasks, each detection task should also remember

[1] We say a node n is the least common ancestor of n_1 and n_2 if n is ancestor of both n_1 and n_2 and no descendant of n is their common ancestor.

where it should get events from (i.e., the probe associated with the corresponding system view instance), the condition that it should check against each observed event (i.e., the transformed timed condition), what variables it will receive from each child detection task, and what variables it should send to its parent detection task. Figure 8 shows a detection task decomposed from a specific version of the signature in figure 7. (Due to space reasons, we cannot show all the detection tasks decomposed from the signature.)

It's worth pointing out that a signature may have multiple workflow trees. For example, 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. In [32], we gave a heuristic approach to generating "good" workflow trees on the basis of the following three principles.

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 n' and there is a path from n_2 to n';
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;
3. Place the nodes for a rare event [2] as close to the leaves as possible.

When there is conflict between the principles, principle 1 has the highest priority and principle 3 the lowest. Please refer to [32] for detailed discussion and the resulting algorithm.

5.4 Cooperative Detection

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 the current version of CARDS but consider it as possible future work.

We maintain several tables to keep events and detection result for each detection task. The *history table*, denoted h, keeps the necessary information of the events on the associated system view instance. The attributes of the history table consist of the timestamps and the event attributes that appear in the assignments or the timed condition of the detection task. For example, if *VictimIP* and *VictimPort* are all the attributes that appear in either the timed condition or assignments in detection task n_1, then the corresponding history table should include *VictimIP*, *VictimPort, begin_time* and *end_time* as

[2] An event is *rare* if it happens infrequently compared with other events

attributes, among which *begin_time* and *end_time* represent the interval-based timestamp.

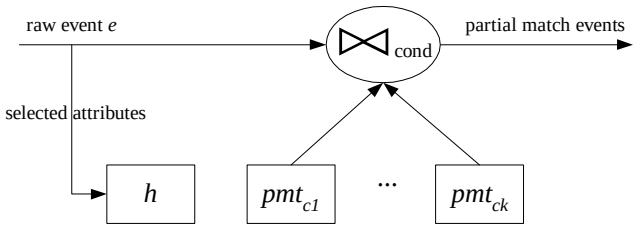
For each child c of the detection task, we maintain a *partial match table*, denoted pmt_c, to keep the variable values assigned by the child detection task. The attributes of pmt_c include all the variables listed under the element `child_partial_match_table` in the corresponding XML description. For example, consider the detection task n_2 in figure 8 and its child detection task n_1, the partial match table pmt_{n_1} has attributes *begin_time_n*1, *end_time_n*1, *var_IP*, and *var_Port*.

If the detection task corresponds to the positive root node in the workflow tree, then a partial match generated by it indicates a potential attack represented by the signature. In this case, the detection task maintains a *matched table*, denoted m, to keep the detection result of the signature. For example, the detection task shown in figure 8 corresponds to the positive root in the workflow tree. Thus, we should keep a matched table and include all the variables involved in the (partial) match (which are specified under the element `partial_match_table`) as attributes. The matched table is used to keep alarms about the specific signature. We will discuss it in further detail later.

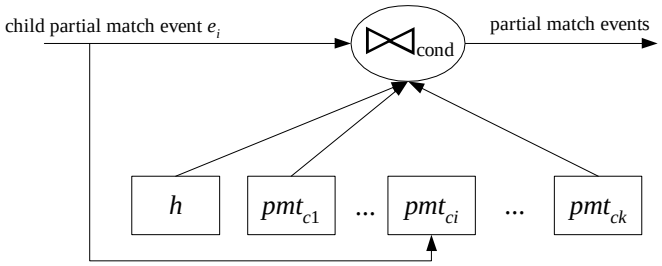
Note that it is generally necessary for a detection task to keep both the event information observed by local probes and the variable values received from its child detection tasks. The detection task may be able to determine that a previously examined event is involved in an attack after receiving additional information from its child detection tasks. Similarly, the detection task needs to examine the information previously received from the child tasks when an event is reported by a local probe. 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.

Each detection task is executed 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 types of events, we call the events on system view instances *raw events* and those sent by a child detection task *partial match events*.

Figure 10(a) shows the processing of raw events. When the detection task receives a raw event e on the associated probe (i.e., the one that provides information on the corresponding system view), it first saves the selected attributes in the history table h. Then it checks whether e satisfies the timed condition along with some tuples in the partial match tables pmt_c. This can be conceptually viewed as a conditional join of e with the partial match tables pmt_c for all child detection tasks c. If matches are found, the detection task



(a) Processing a raw event



(b) Processing a partial match event

Fig. 10. Event processing in a detection task

generates a partial match event for each match. The attributes of the partial match events are specified by the `PMT_attribute` elements, which are either assigned from the e's attribute values or inherited from some child partial match table(s). The newly generated partial match events are then sent to the parent detection task.

As shown in figure 10(b), the processing of partial match events is very similar to that of raw events. When the detection task receives a partial match event e_i from a child detection task c_i, it first saves it in the corresponding partial match table pmt_{c_i}. Then it checks whether this event satisfies the timed condition along with the historical events in h and partial match tables for child c' other than c_i. (This can be conceptually viewed as a conditional join of e_{c_i} with h and the partial match tables for the other child detection tasks.) Similarly, if matches are found, the detection task then generates partial match events and sends them to the parent detection task (if there is one).

As a special case, if the detection task is for a positive root and there are negative nodes in the corresponding workflow tree, the generation of partial match events 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 detection task to work as follows. When the detection task for a positive root node generates partial match events, it saves the result into the matched table m, 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. In other words, we raise an alarm if all of the positive indications of an attack are discovered, and later disable it if counter evidences are found. Although such a solution introduces the possibility of phantom alarms (i.e., alarms that later disappear), it provides better protection than not raising alarms in time and at the same time requires less work for alarm processing.

The detection task for a negative event works in the same way as those for positive events, unless the node is the root of the workflow tree. In the latter case, the generation of partial match events indicates that some previously raised alarms are false, and the corresponding detection task sends the timestamps of the events involved in each partial match to the (detection task of) the positive root, which then removes the corresponding alarms. Note that here we assume the raw events discovered by a probe are uniquely identified by their timestamps.

The execution of detection task can be quite complex if the detection task has many child tasks. The dominant steps are to test the timed condition against the combinations of the newly discovered (or received) event with the tuples in the partial match (or history) tables, which are conceptually conditional joins. A naive processing of a raw event would involve $\Pi_{\forall child\ task\ c}|pmt_c|$ tests of the condition, and similarly, processing a partial match event from the child detection task c would involve $|h| \times \Pi_{\forall\ child\ task\ c' \neq c}|pmt_c|$ condition tests. Such a method corresponds to exhaustive search.

The current version of CARDS uses the aforementioned naive way to process events, since the main purpose is to examine the coordination of different detection tasks. Two approaches can be used to further reduce the execution cost. First, in-memory database query optimization techniques such as in-memory hybrid hash join [11] and T-Tree [39] can greatly reduce the cost of the join operation. Indeed, an in-memory database such as TimesTen [39] 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 $\bowtie_{\forall child\ task\ c} pmt_c$ so that when a raw event is discovered, it can be directly joined with the pre-computed table. Further research is needed to make the execution of a single detection task efficient.

However, we do not address this problem in the current implementation of CARDS, but consider it as future work.

6 Prototype Implementation

We have implemented a prototype of CARDS according to the design discussed earlier. The goal of this version is to examine the feasibility of our approach and understand the issues possibly involved. To focus on the new mechanisms proposed in our work, we made several simplifications to reduce the development time and cost. In particular, we assume that the components of CARDS can successfully establish trust and communicate securely with each other.

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). Xerces Java Parser 1.0.0 is used to process the XML documents representing system views, signatures as well as detection tasks. Since secure communication between the components is not the focus of this system, message transmission between components is carried out over TCP.

In the following, we discuss the implementation issues related to the current prototype.

6.1 Directory Service and DirHelper

Directory service is a critical component for CARDS to achieve scalability. It stores two kinds of information: system view definitions and system view configurations. There are multiple options in implementing the directory service in CARDS: We can use an existing directory service (e.g., Open LDAP directory server, Microsoft Active Directory Service) through standard directory access protocols (e.g., X.500 and LDAP (Lightweight Directory Access Protocol)), or implement a specific directory service for CARDS, or even use a replicated database management system. To make CARDS independent of any specific directory service product, we separate the directory service from the rest of CARDS by introducing a helper component *DirHelper*. That is, each signature manager (or monitor) has a *DirHelper* as a local component, and whenever it needs to access the directory service, it does so by invoking the corresponding methods of *DirHelper*.

Each *DirHelper* provides methods for signature managers or monitors to interact with the directory service, including initializing the directory service,

registering/retrieving/removing system views/probes, listing monitors/probes by system views, and listing all monitors/probes in the directory service.

The current version of *DirHelper* stores the information in a Microsoft SQL Server 7.0 database, which is used as a directory server. Thus, the current version of directory service is certainly not replicated nor scalable. This is to reduce the development time and cost. As shown by our experiments, such a directory service has little impact on the performance for small-scale deployment of CARDS. We plan to replace the database service with replicated directory service (e.g., replicated Open LDAP directory server) that supports LDAP when it is time to examine CARDS in large-scale systems. The replacement of directory server is expected not to affect the rest of the system.

6.2 Signature Manager

Signature manager is further divided into the Graphical User Interface (GUI), the specific signature generator, the (detection) task generator, and the (detection) task distributor. The signature manager accesses the directory through *DirHelper*. The mechanisms of specific signature generator and task generator, i.e., how to generate specific signatures and detection tasks, have been discussed in sections 5.2 and 5.3, respectively. Figure 11 shows a screen shot of the signature manager.

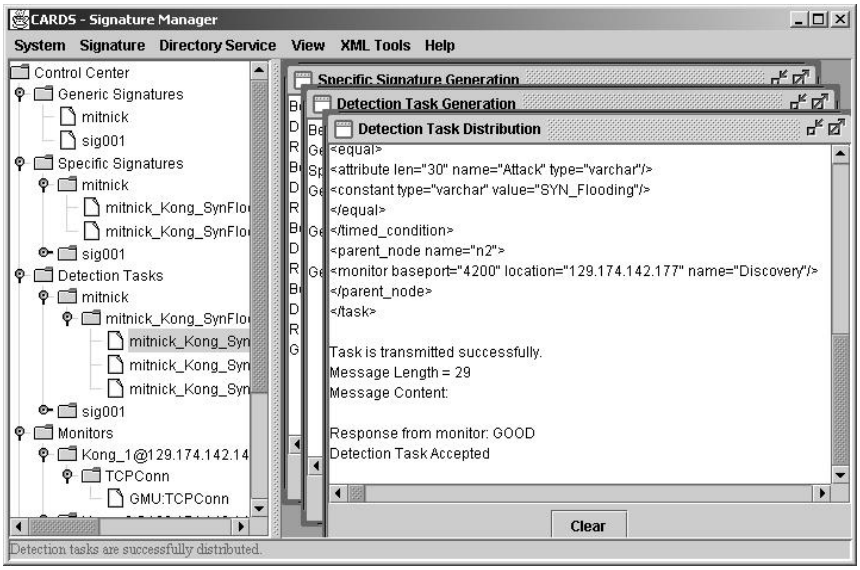


Fig. 11. The screen shot of a signature manager

When generating a specific signature, the specific signature generator retrieves the "contact" information of each monitor involved in the signature. This "contact" information consists of the IP address and a TCP port number, which is called *TaskReceiverPort*. Once started, each monitor will dedicate

a thread, which is called *TaskReceiver*, to listen at the *TaskReceiverPort* for possible incoming task distribution. The same information will be passed to detection tasks when the specific signature is decomposed.

Task distributor distributes the generated detection tasks to the monitors indicated in the tasks, removes obsolete tasks from monitors, or queries the tasks installed in monitors. The protocol between a signature manager and a monitor, which is carried out by the task distributor on the signature manager side and the task receiver on the monitor side, is a one-round request-reply protocol. That is, the signature manager sends a request message to the monitor and receives a response message from the monitor. Each message starts with a length field which indicates the length of the message in bytes (excluding the length field). The rest of the message is a character string as explained as follows.

The message from a signature manager to a monitor consists of three parts: (1) the name of the signature manager, (2) the command, which is one of "ADD", "DELETE", and "QUERY", and (3) the data to be transmitted. The three parts are separated by a special separating character. When the command is "ADD", the data is an XML document representing the detection task. When the command is "DELETE", the data further consists of two parts separated by the special separating character, i.e. a specific signature name followed by a task name. When the command is "QUERY", the data section is empty.

For the "ADD" and "DELETE" commands, the reply from the monitor to the signature manager consists of two sections separated by the special separating character: (1) status and (2) message. Status is either "GOOD" or "BAD", while message is a human readable message (error message when status = "BAD").

For the "QUERY" command, the structure of the reply message consists of the length followed by a list of pairs, each of which consists of a specific signature name and a task name. Again, these logical entities are separated by the special separating character.

6.3 Monitor

Due to its need to process multiple tasks (e.g., monitoring local events, communicating with other monitors and signature managers), monitor is implemented as a multi-threaded application. Each monitor consists of the following threads: the main thread that controls all the other threads, the *TaskReceiver* thread that receives the detection tasks distributed by signature managers, the communication thread that communicates with other monitors, one thread for each active detection task, and one thread for each local probe. The GUI

(i.e., the console in figure 5) and the detection task base are implemented as common modules that can interact with all the threads if necessary. The component Detection Engine shown in figure 5 is virtually implemented as the collection of the active detection tasks. Figure 12 shows a screen shot of a monitor.

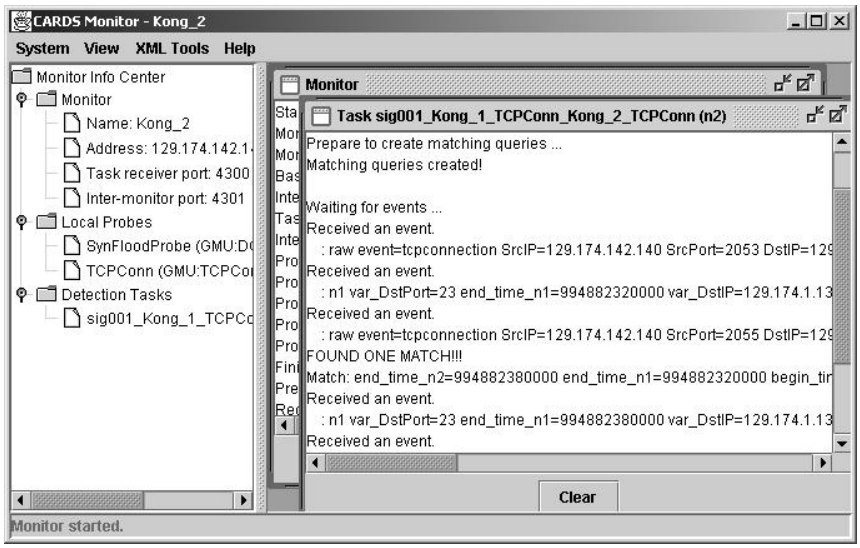


Fig. 12. The screen shot of a monitor

6.3.1 Detection Engine and Detection Tasks

As discussed earlier, detection engine is virtually implemented as the collection of all active detection tasks. A detection task stored in the detection task base is activated when the monitor starts; a detection task distributed to an active monitor is started immediately after it is validated.

Each detection task is indeed an event handler. It processes the raw events generated by local probes or partial match events sent by other detection tasks, which may be located in the same monitor or a remote monitor. The detection task that processes raw events needs to be registered to the corresponding local probes so that the probe will directly dispatch the raw events to it. Similarly, the inter-monitor communication module has access to the list of active detection tasks so that it can dispatch the partial match events, which may be from local detection tasks or remote ones, to the right detection tasks.

Each event is represented as an event message, which is a string consisting of an event type followed by a sequence of name-value pairs. The event type and the name-value pairs are separated with a space. The event type is either "raw", which says the corresponding event is a raw event generated by a probe, or the name of the child detection task, which means that the corresponding event is a partial match event sent by the child detection task. The name-value pairs

describe the attributes of the event. For example, the following is an event message, which gives the attribute values describing a TCP connection.

```
raw event=tcpconnection SrcIP=129.174.142.140 SrcPort=2053
DstIP=129.174.1.13 DstPort=23 begin_time=994882358933
end_time=994882362856
```

Note that the event message format is selected to simplify the prototype implementation. In a deployable implementation, we may have to choose more efficient and interoperable encoding method (e.g., octet string).

To reduce the development cost of this prototype system, we transform the event processing procedures in each detection task into SQL queries and use DBMS (which is Microsoft SQL Server 7.0) to execute them. Our experiments showed that such a choice did have noticeable impact to the performance; however, the performance penalty is acceptable for testing the coordination of multiple CARDS components. Note that the choice of DBMS in the prototype implementation does not imply that our approach has to rely on DBMS or SQL engine to perform the event handling. We plan to replace the DBMS-based event handler with a more efficient one when it is time to improve the performance of each detection task.

6.3.2 Inter-Monitor Communication Module

The monitor relies on the inter-monitor communication module (or simply communication module) to interact with other monitors. More precisely, the local detection tasks of each monitor communicate with the relevant remote detection tasks through the communication module. When a monitor is running, the communication module, which executes as a thread, listens at the inter-monitor communication port called *CommPort* for incoming messages. In addition, the communication module can also be invoked by local detection tasks through method invocation when the local detection tasks need to send messages.

When a local detection task needs to send an event message to another detection task, it submits the message to the local communication module along with the IP address, the port number, the name of the corresponding specific signature and the name of the target detection task through method invocation. If the event message is destined to another local detection task, the communication module will simply pass it to the target detection task by invoking the event receiving method of the local detection task. Otherwise, the communication module will establish a TCP connection with the communication module of the target monitor and transmit the event message along with the names of the specific signature and the detection task.

When the communication module receives a message, it parses the message to find out the names of the specific signature and the detection task. Then it performs a table lookup to locate the corresponding detection task and passes the event message by invoking the event receiving method of the target detection task.

The protocol between monitors (or communication modules of monitors) is quite simple. It is a one-way protocol; each message is encoded as a string, which consists of the name of the specific signature, the name of the target detection task, the event message. The three logical components are separated by end-of-line (EOL) character.

6.3.3 Probes and Probe Loader

Probes observe security relevant data (e.g., audit trails, network traffic) and provide the data in unified forms through system views. They are critical components that make our approach independent of specific data collection mechanisms. The abstraction-based approach in [32] allows probes to either directly collect audit data from the data sources (e.g., network traffic, system call traces) or extract information from the detection of certain signatures via view definition. However, the latter feature is not implemented in the current prototype, but considered as a part of future implementation plan.

Probes are designed as configurable, plug-in modules. Monitor loads the probes through a module named *ProbeLoader*. When a monitor is started, the *ProbeLoader* reads the configuration information of all the available probes, and then locates and loads the corresponding code into the monitor. For example, the following is a piece of configuration information for the probe `SynFloodProbe`. It says the probe provides the system view `SynFloodAttack`, the code for the probe is `edu.gmu.cards.probes.SynFloodProbe`, which can be found in the Java Virtual Machine's class path, and there is no argument to be passed to the probe code.

```
Probe2.name=SynFloodProbe
Probe2.systemView=TCPDOSAttacks
Probe2.className=edu.gmu.cards.probes.SynFloodProbe
Probe2.numOfArgs=0
```

In the current prototype, each probe is dedicated to a specific information source, which collects the event and state information from that information source and reformats the information according to the specific system view(s). We developed several network based probes, including *TCPPacket* (the probe for TCP packets), *LocalTCPConn* (the probe for the TCP connections involving the local host), *TCPConnection* (the probe for all the observable TCP connections), and *SYNFloodProbe* (the probe for SYN flooding attacks). These

probes are implemented by putting a wrapper around the well-known network traffic analysis tool, TCPDump [28]. Thanks to the Windows version of TCP Dump (i.e., WinDump [9]), the probes we developed can run on both Unix platforms and Microsoft Windows.

There are other desirable information sources for which probes should be developed. Examples include host audit trails (e.g., Sun Solaris BSM audit trails [6]), information collected by network management systems and provided through, for example, SNMP or RMON [41], as well as alerts from other security relevant systems (e.g., firewalls and other IDSs). From the perspective of detecting attacks, it is desirable to have more information sources available, since different sources usually have complementary information. Nevertheless, one may not want to deploy all available probes in all the places because of the performance reasons.

6.4 Limitations

One important feature of CARDS is that the intrusion detection task is distributed over all the IDS components that observe the intrusion related data, and these components communicate with each other only when necessary. As a result, the communication cost required for intrusion detection is greatly reduced. We have conducted experiments for a limited number of distributed attacks 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.

The experience with the current version of CARDS also shows several limitations of the approach. Some of the limitations are specific to the current implementation due to the prototype nature of the system and can be addressed easily in future implementation effort, while some others are more essential and require further research effort.

First, when generating specific signatures, CARDS tries to map a generic signature to all combinations of monitors that provide the system view instances underlying the generic signature. This will generate a large number of specific signatures even in a medium scale distributed system. As we discussed earlier, this approach does not consider the relationship between the underlying systems that the monitors are protecting, and not all of the specific signatures correspond to attacks that may cause severe damages. In reality, we may choose to detect some of the possible attacks and tolerate others due to some administrative concerns. Future work is needed to address this issue.

Second, the experience with CARDS revealed another issue about time other

than the clock discrepancy problem. Even if the clocks of all the component systems in a distributed system are well synchronized, there are still problems involving time because of the network latency and multi-process nature of the contemporary operating systems. For example, the same TCP connection will not be labeled with the exact same timestamp on the two end-point computers that are far enough from each other. Even if the network latency is negligible, the timestamps may still be different if the workload of the two computers are very different. These observations imply that we have to properly deal with the timestamps of the events.

A simple countermeasure is to set up a threshold t as the maximum time difference and handle timestamps in a relaxed way. For example, 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.

Third, the abstraction-based approach certainly introduces overhead in processing the events observed from target systems, though it can hide the difference between heterogeneous platforms. The overhead comes from the preprocessing of the events, which normally involves reformatting the events. Sometimes, there may not be direct correspondence between events observed on different platforms, and a probe may need to perform aggregation and other simple event processing before it can present an event on a system view. The hierarchical abstraction framework, which was proposed as a part of the abstraction-based approach in [32], gives a partial solution to this problem, though it is not implemented in the current version of CARDS.

Fourth, the communication between CARDS components is based on the TCP protocol. Though we did not experience any problem during the experiments, we realize the possibility that the CARDS components may not be able to communicate with each other as in the normal situation if the system is under attacks from some powerful enemies. This is actually a realistic problem for all the distributed IDSs. An ideal solution is to provide some special communication channels between IDS components; however, this is the same as requiring a separate network for IDS components, and this very separate network may be subject to attacks as well. Another approach is to let IDS components rely on simpler communication mechanisms such as UDP. However, the reliability of the message transmission must be taken care of as well.

A closely related issue is how to perform intrusion detection if some of the component IDSs or the communication channels are not reliable. The current version of CARDS may miss critical attacks or generate false alarms if the above situations happen. Indeed, this problem is not unique to CARDS, but common to all distributed systems. Additional research is required to address

this problem.

Finally, with our current approach the IDS has to wait for the completion of the observed events before it can detect any attack. For example, if a signature involves a TCP connection, the corresponding event will not be generated until the connection is closed or broken. Indeed, this implies that an attack cannot be detected until after all the positive events involved in the attack complete. A counter measure is to let the probe to report the starting and the ending point of an event separately. Although the IDS may not always be able to detect the attack if the ending times of certain positive events are critical to the existence of the attack, the IDS can often reason about the attack at an early stage. However, additional research is needed to implement this method efficiently.

7 Conclusion and Future Work

This paper described the design and implementation of a decentralized research prototype IDS named CARDS, which aims at detecting distributed attacks that cannot be detected using data collected in any single place. CARDS is based on the abstraction-based techniques proposed in [32]; it has two distinguished features. First, the 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, and then mapped to specific systems that need to be protected. Second, CARDS decomposes the specific representation of distributed attacks into detection tasks, which are distributed to and cooperatively executed by the component IDSs located at different places.

In our future work, we will continue to refine the implementation of CARDS and seek solutions to address its limitations. In particular, we would like to improve the performance of each single detection task, enable component IDSs to tolerate clock differences, and develop solutions to help IDS survive unreliable communication channels as well as unreliable component systems.

Acknowledgement

The authors are grateful to Professor Ray Hunt for his valuable suggestions.

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] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation 10-February-1998, February 1998.

[6] *Solaris SHIELD Basic Security Module Revision A.*

[7] 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.

[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] L. Degioanni, F. Risso, and P. Viano. WinDump: tcpdump for windows. http://netgroup-serv.polito.it/windump/.

[10] D. E. Denning. An intrusion-detection model. In *Proceedings of 1986 IEEE Symposium on Security and Privacy*, pages 118–131, Oakland, CA, May 1986.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] A. K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 259–267, December 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, 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] S. McCanne, C. Leres, and V. Jacobson. Tcpdump. Lawrence Berkeley National Laboratory, ftp://ftp.ee.lbl.gov/tcpdump.tar.Z.

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

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

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

[32] P. Ning, S. Jajodia, and X.S. Wang. Abstraction-based intrusion detection in distributed environments. To appear in ACM Transactions of Information Systems Security (TISSEC). Available at http://www.csc.ncsu.edu/faculty/ning/pubs/AbstractID.ps, February 2001.

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

[34] 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.

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

[36] 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.

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

[38] 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.

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

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

[41] W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addision Wesley, 1999.

[42] 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.

[43] 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.

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

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

[46] 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.