Collaborative Intrusion Detection System (CIDS): A Framework for Accurate and Efficient IDS

Yu-Sung Wu, Bingrui Foo, Yongguo Mei, Saurabh Bagchi
School of Electrical and Computer Engineering, Purdue University
Email: {yswu,foob,ymei,sbagchi}@purdue.edu

Abstract

In this paper, we present the design and implementation of a Collaborative Intrusion Detection System (CIDS) for accurate and efficient intrusion detection in a distributed system. CIDS employs multiple specialized detectors at the different layers – network, kernel and application – and a manager based framework for aggregating the alarms from the different detectors to provide a combined alarm for an intrusion. The premise is that a carefully designed and configured CIDS can increase the accuracy of detection compared to individual detectors, without a substantial degradation in performance. In order to validate the premise, we present the design and implementation of a CIDS which employs Snort, Libsafe, and a new kernel level IDS called Sysmon. The manager has a graph-based and a Bayesian network based aggregation method for combining the alarms to finally come up with a decision about the intrusion. The system is evaluated using a web-based electronic store front application and under three different classes of attacks – buffer overflow, flooding and script-based attacks. The results show performance degradations compared to no detection of 3.9% and 6.3% under normal workload and a buffer overflow attack respectively. The experiments to evaluate the accuracy of the system show that the normal workload generates false alarms for Snort and the elementary detectors produce missed alarms. CIDS does not flag the false alarm and reduces the incidence of missed alarms to 1 of the 7 cases. CIDS can also be used to measure the propagation time of an intrusion which is useful in choosing an appropriate response strategy.

Keywords: *Intrusion detection, Event correlation, Bayesian network based detection, False alarms, Missed alarms.*

1. Introduction

Many critical parts of our information infrastructure comprise distributed computer systems with myriad application level and system level components deployed on multiple platforms. The infrastructures are vulnerable to attacks, especially when they have an open, connected architecture with interactions with untrusted clients over untrusted networks. Intrusion detection systems (IDSs) are deployed to protect the computer infrastructures. The classical IDSs fall into two classes – *anomaly based*, and *misuse based*. An anomaly based IDS specifies the normal behavior of users or applications and considers any pattern falling outside the defined behavior as an attack. A misuse based IDS specifies the signatures of attacks and parses audit files to detect any matches. The metrics for evaluating an IDS are false alarms (or, false positives), and missed alarms (or, false negatives). Individual IDSs are often found to be unsatisfactory with respect to either or both of the metrics. For instance, anomaly based detection can generate many false positives since deviation from the specified normal behavior is not necessarily an attack. Also, if the definition of normal behavior is updated at runtime, an expert intruder can slowly change her behavior to finally include it in the definition. This would then give rise to a false negative. Misuse based detection can generate many missed alarms since for most practical open systems it is very difficult to define an exhaustive attack data base. Also current misuse based IDS products generate false alarms as our experience with Snort reported here also shows.

In this paper we propose a system model that employs multiple specialized detectors installed in different layers of the system, and a management infrastructure for collating the alerts from the multiple elementary detectors and synthesizing a global and aggregate alarm. For this purpose, a system is divided into the network layer, the kernel layer and the application layer. We claim that the aggregate alarm is more accurate than the elementary alarms, i.e., it reduces the incidence of false alarms and missed alarms. The system should also be efficient in that the performance degradation compared to the baseline case of no detector, or of a single detector, should not be substantial. We design and implement a system called the Collaborative Intrusion Detection System (CIDS) to demonstrate the feasibility of the idea.

CIDS employs three elementary detectors (EDs)–*Snort*, a network level detector, *Libsafe*, an application level detector, and a new detector called *Sysmon* that executes at the kernel level. CIDS has a manager to which the alerts from the EDs are communicated. Sysmon consists of a modified Linux kernel for intercepting certain OS activities — file access, and illegal signals. The EDs may be monitoring different system components, possibly on different hosts and communicate with the manager through a message queue (MQ). The MQ design enables detectors on different hosts to communicate securely with the manager.

Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)
1063-9527/03 $17.00 © 2003 **IEEE**

The CIDS manager consists of a Translation Engine, an Inference Engine, a collection of Rule Objects and a Response Engine. The Translation Engine translates the alerts from the different EDs into a common format and attaches identifying information, such as the host ID from which the alert originated. The Inference Engine uses the rule base to calculate the probability of an attack for each class and flags an alarm if a probability exceeds a threshold. The Inference Engine accommodates the choice of different techniques for rule matching. In the current system, there is a graph-based inference engine and a Bayesian network based inference engine. Both are able to handle partial matches of the observed events with the rule base events and non-determinism in the order in which the events are received. This design decision is guided by the practical observation that such partial streams and order non-determinism are common in distributed systems. The Response Engine can take various responses depending on the attack type and its characteristics (such as, the propagation speed). The current system uses the simple response of terminating the connection which originated the suspect packets.

CIDS is evaluated with respect to performance degradation and accuracy of detection. Attacks from three different classes are used to stress the system – buffer overflow attack, flooding attack and script based attack. An electronic store front running on an Apache web server, implemented using CGI Perl scripts, and accessed using a web browser client is used as the workload. The application allows the typical operations of creating a user profile, browsing the catalog, adding items to a cart and completing an order, with multiple operations being grouped to form a transaction. The performance is measured by the number of transactions per second (tps) and the CPU available to the web server. The performance degradation is given by the values of these metrics in CIDS compared with the baseline system configuration with no detector. The results show degradations of 3.9% and 6.3% under normal workload and a buffer overflow attack respectively. Experiments are then conducted to explore the cases of false alarms and missed alarms. The false alarm experiment is conducted with the normal transaction and 3 variants. The missed alarm experiment is conducted with 7 different attack types corresponding to the 3 attack classes. The results show that the normal workload generates false alarms for Snort and Sysmon, and missed alarms for Snort (3 of 7 cases), Libsafe (6 of 7 cases), and Sysmon's two configurations (3 of 7,and 4 of 7 cases). CIDS does not show any false alarm and reduces the incidence of missed alarms to 1 of the 7 cases. The third set of experiments tracks the timing information for detection by the EDs and inference at the manager. This brings out the speed of propagation of the attacks and the latency of each step in a CIDS workflow.

The work presented here has the following claims of innovation – (i) it proposes a new system model for correlating alerts from multiple elementary detectors to perform more accurate intrusion detection; (ii) it presents an incremental inference engine capable of tolerating non determinism observed in practical systems; (iii) it provides an intrusion detection framework in a distributed multi-host system; (iv) it presents a new kernel level detector and shows it to be effective under several attack scenarios; (v) it provides timing analysis of events which can aid in selecting an appropriate intrusion response.

The rest of the paper is organized as follows. Section 2 refers to related research. Section 3 presents the architecture of CIDS and describes its components. Section 4 provides an instantiation of the architecture and presents the specific configuration of the system being evaluated in this paper. Section 5 describes the experiments and the results. Section 6 concludes the paper with mention of future work.

2. Related Research

Several researchers have addressed the problem of false alarms and missed alarms with traditional IDSs which are classified as anomaly-based and misuse-based [8]. Also, traditional IDSs often generate a very large number of alerts for practical attack scenarios. The alarms correspond to elementary goals of the attack being realized. This large volume of alarms makes it difficult for a system administrator or even an automated intrusion response system to take appropriate actions. To counteract this problem, several researchers have developed alert correlation methods to construct attack scenarios. One class of techniques ([16],[19]) combines alerts based on similarity of certain alert attributes. For example, in [16], source and destination IP addresses and ports are used for determining similarity and graphs are drawn with links between related alerts. However, this class misses out on correlating a large set of related alerts. A second class of techniques [2],[6] use training set data to determine relations between alerts. In [2], attacks are characterized by pre-condition, post-condition, attacker actions, detection actions, and verification actions to determine if the attack succeeded. Knowing these attributes, they provide techniques to correlate alerts. However, the challenge remains to determine the attack characteristics. The most promising approach in alert correlation is demonstrated by [3],[13],[18] which correlate alerts based on pre-conditions and post-conditions. Two alerts are correlated if the precondition of a later attack is satisfied by the post-condition of an earlier attack. This volume of work addresses a related but distinct problem than our work. The goal is to cluster the alerts corresponding to the distinct elementary attacks that form part of a larger attack. Our goal is to increase the accuracy of detection of each elementary attack. Their work uses detectors for detecting different types of attacks while we can have multiple detectors that detect the same kind of attack and use the multiple detection to increase the assurance in the alert. Thus, our work can be considered complementary and benefit

their work. Consider that the elementary alerts that they consider are not the alerts from individual detectors, but alerts from our CIDS manager. One concern about this volume of work is their ability to deal with non-determinism in alerts from detectors in real-world intrusion situations, such as, missed alarms, alarms appearing in different orders. For example, in [3], it would be difficult to detect multiple missing alerts and there is no notion of assurance which is reduced as the number of missing alerts increases. In none of this work is it obvious how overlapping alarms corresponding to multiple concurrent attacks on multiple hosts will be handled. On the other hand, CIDS can be deployed as a distributed intrusion detection system.

3. Architecture of CIDS

In this section, we describe the architecture of CIDS which has the following components – the Elementary Detectors (EDs), the Message Queue (MQ), the Connection Tracker, the Manager, and the Response Engine.

3.1. Elementary Detectors

The Elementary Detectors are the specialized intrusion detectors that are distributed through the system. From an architecture standpoint, each host is divided into three layers– network layer, kernel layer and application layer. The network layer consists of the network protocol stack. The kernel layer consists of the operating system and its managed services. The application layer consists of everything else running as software on the host, including middleware. The EDs and the manager can be located on different hosts and communicate through a generalized Message Queue structure which enables communication regardless of where the communicating processes are located. A possible system view is represented in Figure 1.

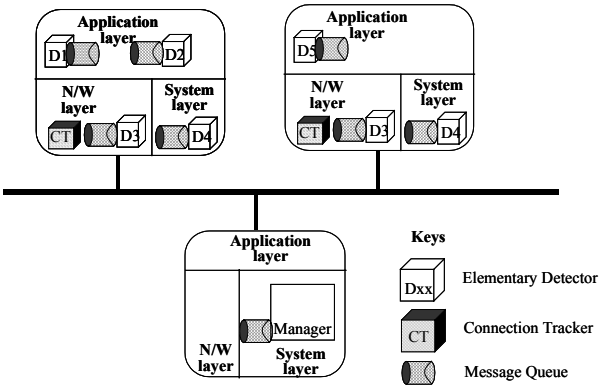


Figure 1. System view of Elementary Detectors and Manager

The EDs may be off-the-shelf detectors. CIDS does not mandate substantial changes of such off-the-shelf detectors. The only change is in the function when an alert is generated by the detector which puts an alert out on the MQ destined for the manager.

Different hosts can have different configurations of the EDs. This is an important design principle since in a distributed system with heterogeneous services, different services on different hosts may need different kinds of detectors. There may be EDs specialized for detecting different kinds of attacks, EDs with overlapping functionality, and EDs which are an integral part of an application (such as, a Voice-over-IP application that has inbuilt rules to detect traffic fraud attacks). The rule base at the manager has to be initialized with values for confidence placed on the alarms from the individual detectors for different types of attacks. The system will exhibit a faster learning curve if the initial confidence values are based on the specialized functionality of the detector. For example, an alarm from Libsafe which is a buffer overflow detector should have a higher confidence for the Rule Object for buffer overflow, rather than for flooding.

3.2. Message Queue

The components in the system communicate with one another using the Message Queue (MQ). MQ uses TCP as the data transport. Each message has a per sender unique monotonically increasing serial number and a signature which is the SHA1 hash value of the message body, the serial number, and a secret key. Each ED in the system has a secret key that is shared with the manager. Thus, without knowing the secret key, there is no way to forge a message or replay a legitimate message.

3.3. Connection Tracker

The Connection Tracker is a kernel level entity which maintains the mapping of port number to process ID of the process which has an active connection on the port. For this, it intercepts the system calls for accepting incoming connections and terminating connections. The manager may query the Connection Tracker to generate information about the target for which an alert is raised. The Connection Tracker maintains the information in a queue data structure.

3.4. Manager

The manager is the workhorse and the key differentiating component in CIDS. The manager is responsible for aggregating the information from the different detectors and making a combined system-wide decision about the existence of an intrusion. The architecture permits the manager to monitor multiple hosts. There is one manager for the administrative domain on which you would want to correlate alerts.

The manager has the following components: Translation Engine (translates the alert from an ED into a CIDS understandable abstract event form), Event Dispatcher (dispatches the event to the appropriate host's Inference Engine instance), Inference Engine (matches the received events against the Rule Objects to come up with a determination of intrusion), and Combining Engine (collates the decisions from the different instances of the Inference Engine and decides on the appropriate response). An architecture of the manager with the different components is shown in Figure 2.

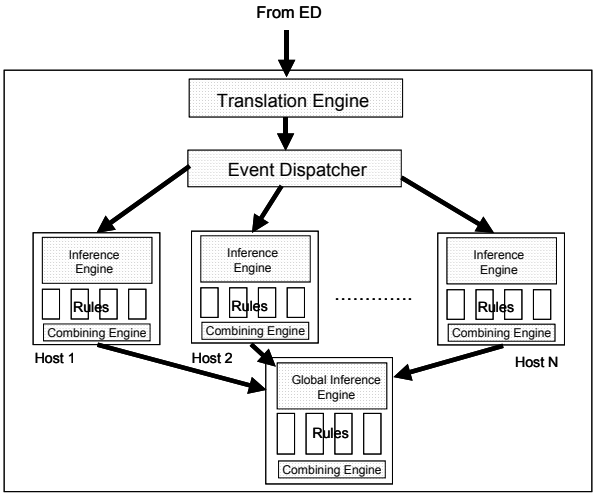


Figure 2. Architecture of the CIDS Manager

3.4.1. Translation Engine (TransEng). The Translation Engine (*TransEng*) translates the alert from the ED into a common format. The format is as follows.

- Event Id (EID): ID of the detector (DID); Specific ID given to the alarm type by the detector (AID)
- Location Info (LID): Source IP address and port number (SIP, SPN); Destination IP address and port number (DIP, DPN), Process ID (PID)

Each detector in the system has a unique ID which is filled in as DID when it sends an alarm. In addition, an ED may specify a unique ID specific to its detection, e.g., an ID corresponding to the nature of the attack detected. This may be used by the Inference Engine in deciding on the Assurance Value for the attack, e.g., an alarm corresponding to a buffer overflow by Snort will contribute less to the Assurance Value of a rule for detecting flooding attacks. The PID in the Location Info field gives the id of the process to which the suspected malicious packet was destined. The responsibility for filling in these fields is shared between the ED and the manager. TransEng fills in DIP, since the MQ maintains information about the IP address from which the message came. When Snort raises an alarm, it fills in the DPN field while TransEng fills in the PID field by querying the ConnectionTracker. Libsafe on the other hand fills in PID and TransEng queries Connection Tracker to fill in DPN.

3.4.2. Event Dispatcher (EvDis). The Event Dispatcher (*EvDis*) dispatches the events to the Local Inference Engine of the host corresponding to the destination (the DIP field) of the event. The events being forwarded by EvDis are maintained in an Inference Queue in time order at each Local Inference Engine. The events are logically grouped by the target process they correspond to (the PID field). The Inference Engine can then process the events efficiently to determine if there is an attack on the target process. The logical structure of the Inference Queue is shown in Figure 3.

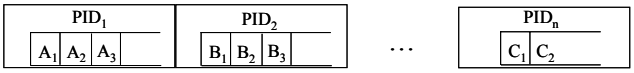


Figure 3. Logical structure of Inference Queue at each host's Inference Engine. A_i, B_i, C_i are events.

3.4.3. Inference Engine. The goal of the Inference Engine is to process the events in the Inference Queue and come up with the determination if an intrusion is in progress. The determination is quantified by an Assurance Value. The matching of the observed events is performed against a rule base consisting of Rule Objects. There is an instantiation of the Inference Engine and the Rule Base for each host monitored by the manager and a global Inference Engine and Rule Base which processes the output from the local engines. The manager's rule-base is maintained by a system administrator and as part of future work, we plan to provide a feedback loop that lets the rules be updated based on the success of the detection.

We describe below the design of the Inference Engine without the need to make a distinction between a Local and a Global Inference Engine. The distinction arises due to the specific rules that are in their corresponding rule bases. We have designed two different kinds of Inference Engines – a Graph-based Inference Engine and a Bayesian Network based Inference Engine.

Graph-based Inference Engine

Each Rule Object is a graph with the nodes being the events associated with an attack type and the edges denoting a sequencing of the events and marked with the confidence associated with the sequence. Intuitively, the Assurance Value (AV) for a particular attack is given by the sum of the edges in the longest path of the entire sequence of observed events. A path is considered longer if it has a higher assurance value. When a new event is added to the Inference Queue, the Inference Engine checks to see if it is *fusionable* with the events being currently matched. Two events are considered fusionable if they can be events in a common attack instance. This is application specific and can be customized in the environment. Currently, if two events are for the same target process they are considered fusionable. A sequence of fusionable events forms an *event stream*.

The Inference Engine matches different streams of events that are in its Inference Queue in parallel. For each stream, it

matches it against each Rule Object. Let us consider below the processing of events from a single event stream being matched against one Rule Object graph. The Inference Engine drains an event from the Inference Queue and if fusionable with the previous event, puts it in a Match Queue. Each element in the Match Queue contains the highest Assurance Value associated with a path having the itself as the terminal node, and the predecessor event in such a path.

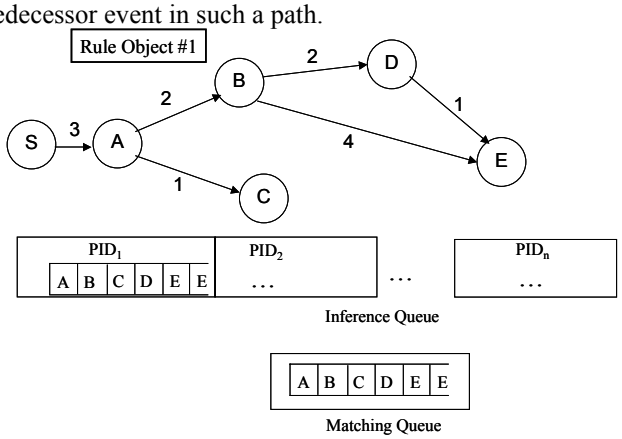


Figure 4. Example of Graph-based Rule Object and Event Stream

An event would expire from the Match Queue when it is determined that the fresh alerts are corresponding to a different attack. This can be determined by using metrics such as the source of the attack, the time interval, the target process [1].

Let us now consider a running example of a Rule Object graph and the processing at the Inference Engine. The Rule Object is shown in Figure 4. The nodes correspond to possible events with S being a special node corresponding to the start event. The events observed in time order are A;B;C;D;E;E. The snapshots of the Matching Queue on receipt of each event are shown in Figure 5. Only the arc corresponding to the newly added event is shown. The assurance value for an added event may not be monotonic as the addition of C shows.

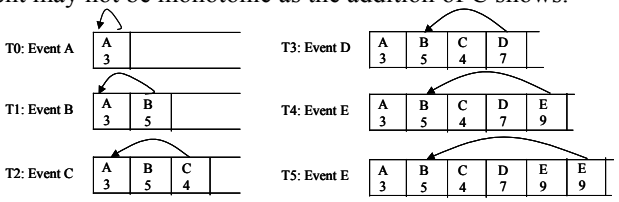


Figure 5. Matching Queue snapshots on receiving each event in event stream

The Inference Engine processes an event by scanning through the possible predecessors of the event in the Rule Object and calculating the Assurance Value for the new event. The Assurance Value at the end of the processing is converted to a probability value for matching against thresholds by dividing it by the maximum possible Assurance Value from the Rule Object. The algorithm runs in $O(V^2+VE)$ where V is the number of events being processed and E the number of edges in the Rule Object graph. The algorithm is given in Figure 6.

```
G_i := Rule Object Graph i
M := Matching Queue
T := Tail of the Matching Queue
BestAV := 0
BestPredecessor := T
AddEvent ( NewEvent )
{
  if (NewEvent.Fusionable(M[T].event) M.insert(NewEvent);
  for k = T-1 to 0
  {
        if ( !ExistEdge(G_i(M[k].event , NewEvent) ))
          continue;
        NewAV := EdgeValue(G_i (M[k].event , NewEvent)) + M[k].AV;
        if ( NewAV > BestAV) {
          BestAV := NewAV;
          BestPredecessor := k;
        }
  }

  M[T].AV = BestAV
  M[T].Predecessor = BestPredecessor
}
Probabilistic AV = BestAV/MaximumAV
if (Change in Probabilistic AV) send message to Combining Engine
```

Figure 6. Algorithm for processing a new event at the Graph-based Inference Engine

How to handle missing events? In a practical system, there will be missing events and exact matches with the Rule Object graph will not always be possible. In order to handle this condition, we allow partial matches with a discounted Assurance Value. The discounted Assurance Value is obtained by multiplying the Assurance Value by a discount factor, which is given by the number of observed event nodes divided by the total number of nodes on the path. Thus, if event B was missing and the event stream was A;D, the Assurance Value would be 2/3*(3+2+2)=4.67.

Parallelization and Distribution. The Inference Engines can be distributed very intuitively by having each local Inference Engine execute on a separate host. Each Inference Engine can parallelize the processing by matching each separate event stream against a Rule Object graph concurrently.

Bayesian Network-based Inference Engine

Bayesian Network is a compact representation of joint probability distributions via conditional independence [12]. In a Bayesian Network, the nodes represent random variables and edges the direct influence of one variable on another. A set of conditional probability distributions is associated with each node and a node is considered conditionally independent of its ancestors given its parents. There are two steps to modeling a Bayesian Network. The first step is creating the graph which describes the conditional probability relationship among events by putting an edge from event A to event B if B is conditioned on A. Next, we have to specify all the conditional probabilities, i.e. P(B|A).

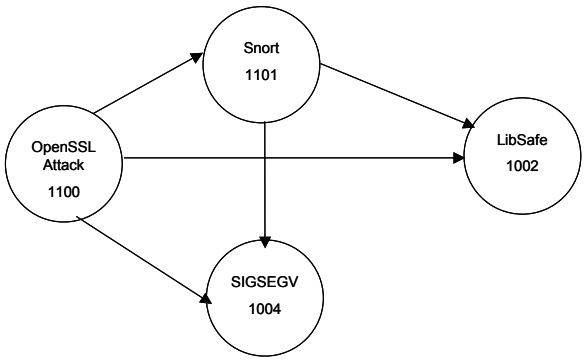


Figure 7. Example of Bayesian Network based rule for OpenSSL attack type

In CIDS, we model the rules in a very similar way. For example, in the OpenSSL attack case, we have the Bayesian Network shown in Figure 7 as the BN-based rule in the manager. The conditional relationship is put in a causal way. For instance, we know that a OpenSSL attack could cause a Snort alert so we put an edge from the 'OpenSSL Attack' event node to the 'Snort' node. After this, we give all the conditional probabilities P(Snort | OpenSSL Attack), P(Libsafe |OpenSSL Attack, Snort), P(SIGSEGV | OpenSSL Attack, Snort).

When a new alert comes into the Inference Queue, it is checked for the fusionable property and all fusionable alerts are grouped into a vortex. The Matlab Bayesian Network Toolbox [11] is then invoked and events in the vortex are fed to it as evidence nodes. The inference function of the toolbox is executed to acquire the probability of the root node (e.g. the Open SSL Attack node), which is the alert probability of the attack based on the observed evidences.

Role of Global Inference Engine. Alerts determined by a Local Inference Engine are fed into the Inference Queue of the Global Inference Engine (GIE). The motivation for a GIE is that a Local Inference Engine can determine if an individual service is under attack, but a GIE can determine if the aggregate distributed service is under attack. For example, the authentication information in an electronic store front may be compromised by launching a parallel attack against the DNS service (to misdirect traffic to a rogue host) and the SSH server (trying to sniff authentication information being exchanged between the client and the server).

How Rule Objects are created. Rule objects are created individually for each attack type and built up incrementally for the entire system. Future work will be on retuning the confidence in a detector based on feedback from the response phase to decide if detection was successful or not.

3.4.4. Combining Engine. The Inference Engine matches the event stream against the different Rule Objects using both the graph-based and Bayesian network based approaches. The Probabilistic Assurance Value from a particular type of Inference Engine is the maximum of the values from the different rules. The Combining Engine takes the greater of the Assurance Values from the two approaches. It compares this against a set of threshold values and depending on the threshold that is exceeded, the appropriate signal is sent to the Response Engine. The Assurance Value staying below the lowest threshold indicates that the alerts generated were false alarms and CIDS disregards them.

3.5. Estimating Speed of Attack Propagation

A key determinant of the response for an intrusion is the relative time taken to deploy the mechanism and the speed of propagation of the intrusion. A benefit of the CIDS architecture is that the propagation speed can be estimated using the timing of the alerts from the EDs. Information about cascaded security vulnerabilities in communicating services can be obtained by various tools developed by researchers such as Kaaniche [4], Deswarte [5], and Dacier [14]. From such information, a graph of services can be created and the time for a service to be affected can be extrapolated from an estimate of the speed of propagation of the intrusion. Consider an attack to service S1 has been determined by the CIDS manager at time t1 and a second attack to service S2 has been determined at time t2 (t2 > t1). If services S1, S2 and S3 are placed in a linear chain, then an unweighted linear extrapolation will indicate S3 will get affected by the intrusion at t2+(t2-t1) and therefore, the response mechanism must be able to complete by then.

4. Instantiation of CIDS

We build a system that instantiates the CIDS architecture and is described in this section with all the components that are used in the experimental evaluation. CIDS is currently implemented on Red Hat Linux 8.0. The main components of the current system are: Manager, Three EDs, Response Engine, Netfilter, and Apache web server as workload. A schematic of the system is shown in Figure 8.

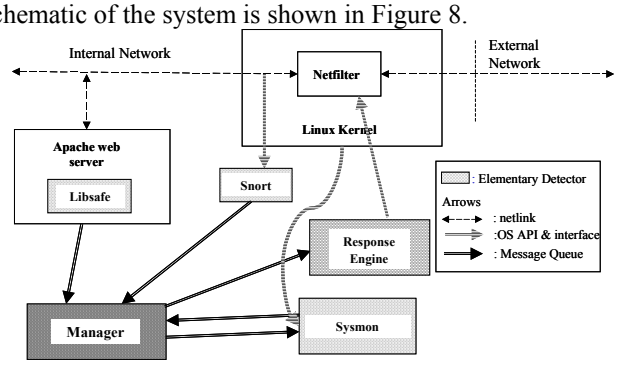
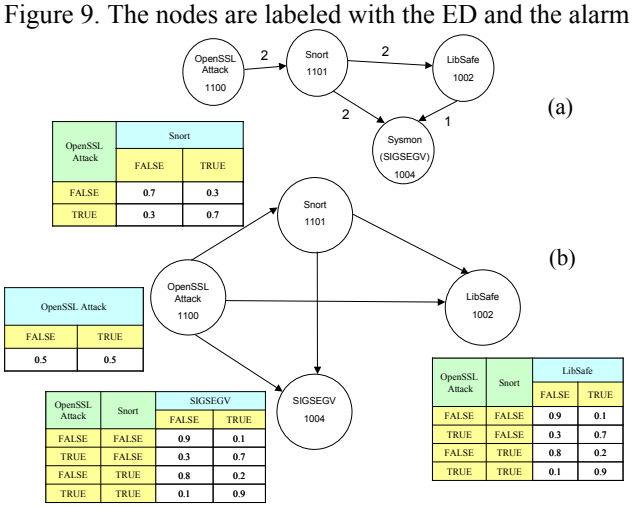


Figure 8. Currently implemented CIDS

4.1. Manager

The manager has the Graph-based and the Bayesian Network based Inference Engines. The Inference Engines use Rule Objects for each attack type. Examples of graph and Bayesian Network rules for the Open SSL attack are shown in Figure 9. The nodes are labeled with the ED and the alarm ID.



OpenSSL Attack	Snort	
	FALSE	TRUE
FALSE	0.7	0.3
TRUE	0.3	0.7

OpenSSL Attack
FALSE	TRUE
0.5	0.5

OpenSSL Attack	Snort	SIGSEGV	
		FALSE	TRUE
FALSE	FALSE	0.9	0.1
TRUE	FALSE	0.3	0.7
FALSE	TRUE	0.8	0.2
TRUE	TRUE	0.1	0.9

OpenSSL Attack	Snort	LibSafe	
		FALSE	TRUE
FALSE	FALSE	0.9	0.1
TRUE	FALSE	0.3	0.7
FALSE	TRUE	0.8	0.2
TRUE	TRUE	0.1	0.9

Figure 9. (a) Graph-based and (b) Bayesian Network-based rules for the OpenSSL attack

4.2. Elementary Detectors

Three EDs are used in CIDS: *Snort*, *Libsafe* and *Sysmon*, of which Sysmon is a new detector that is designed and implemented for CIDS.

Snort [15] is a network level intrusion detector which sniffs the network flow by using *libpcap* [10]. Snort is capable of sniffing TCP, UDP, and ICMP packets. It then compares the sniffed packet against its rulebase and signals alerts when it matches rules. The snort rule has a very versatile description format and the Snort engine can perform simple matching by port numbers or IP addresses or perform more involved pattern matching in the payload of a packet and stateful protocol matching. We use Snort version 1.9.0 with the complete default set of rules (snort.conf version 1.124).

Libsafe [9] is an application level ED. It provides a middleware software layer that intercepts function calls made to a set of C library functions that are known to be vulnerable to buffer overflow attacks, such as string manipulation routines. A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. This prevents attackers from 'smashing' (overwriting) the return address and hijacking the control flow of a running program. Libsafe is limited to protecting against stack buffer overflow and is unable to catch heap or static buffer overflows. Also, according to our experiment, it misses some stack buffer overflow situations with optimized program code when it fails to get an accurate estimate of the stack frame size. We use Libsafe version 2.0-1 for our system.

Sysmon is a new kernel level detector which comprises modifications to the Linux kernel for intercepting certain OS activities. Sysmon has two functions.

1. *Monitoring file accesses by intercepting the sys_open and sys_execve system calls*

   Sysmon monitors all the file accesses. Currently, we only put simple access rules for conceptual purpose. The rules we have are: (a) Not allowing access to other users' home directories; (b) Not allowing the execution of commands 'ls', 'rm', and 'gcc' to prevent the hacker from listing or removing files and from compiling malicious codes on the machine. A more comprehensive approach in our current plans is to create a list of allowed file accesses based on audit data.

2. *Intercept interested signals*

   Sysmon can intercept all signals that a monitored process is receiving. Currently, we only process the SIGSEGV segmentation fault signal. This signal is usually a result of unsuccessful return address overwriting or unsuccessful injection of malicious code.

4.3. Apache Web Server Workload

The Apache web server version 1.3.24 is used as the workload. An electronic store front is implemented using Perl CGI scripts which consists of 3 major parts: (i) Registering a user profile or account, (ii) Browsing the online catalog and placing items in a shopping cart, and (iii) Completing the order.

5. Experiment

In this section, we describe the experiments used to evaluate CIDS. We describe the normal workload, the simulated attacks, the performance measurements, the evaluation of false alarms and missed alarms, and the timing measurements.

5.1. Electronic Store Front Workload

The normal workload used in the evaluation of CIDS is a client transaction which exercises different functionalities of the web-based electronic store front. The client transaction is written in HTML 1.1 and consists of the following steps.

1. Getting the html page that allows a customer to register a profile.
2. Sending information to *mailer.cgi* to create a profile.
3. Getting the html page that contains the store catalog.
4. Sending information to *cart.cgi* to place an item in the shopping cart.
5. Viewing the shopping cart by executing *view_cart.cgi*.

6. Viewing the checkout information by executing *checkout.cgi*.
7. Sending information to *complete.cgi* to complete the checkout.

A basic TCP stream socket client program is used to send and receive all the steps synchronously and in sequence with no delay between successive steps.

5.2. Attack Types

We simulate three classes of attacks to evaluate CIDS. For each class, we develop multiple attack types and variants for some of the types. A detailed description of the attack types is not relevant to the evaluation and hence only a high-level overview is given along with references to their details. A problem we faced in the study was availability of code to simulate the attacks. We developed the attack code from scratch for most types and occasionally got fragments of code that we could modify and use. An important design principle for the experiments was to separate the developer of the attack code from the designers of the system to prevent biasing of the attack methodology either to favor CIDS, or exploit known vulnerabilities in CIDS. The two groups had no communication during the entire length of the study.

1. Buffer Overflow Attack. A buffer overflow attack exploits the fact that oftentimes programs do not check for boundary conditions in operations such as accessing arrays. This can be used to overwrite parts of the stack such as the return address and cause malicious code to be executed.

1.1. *Apache Chunk attack* [1]. Versions of the Apache web server up to and including 1.3.24 and 2.0 up to and including 2.0.36 contain a bug in the routines which deal with invalid requests which are encoded using chunked encoding. This bug can be triggered remotely by sending a carefully crafted invalid request. In 32-bit platforms (including ours), the attack causes a stack buffer overflow and process crash, while in 64-bit platforms, it could be used to execute malicious code.

1.2. *Open SSL attack* [16]. This is a static buffer overflow attack and contains a remotely exploitable buffer overflow vulnerability in OpenSSL servers prior to 0.9.6e and pre-release version 0.9.7-beta2. This vulnerability can be exploited by a client using a malformed key during the handshake process with an SSL server connection using the SSLv2 communication process. As is previously documented, in our version of Linux (Redhat 8.0) and Apache web server (1.3.24), this attack can cause a segmentation fault, but not execution of malicious code. This attack consists of two phases —determining the version of Apache web server to determine the memory layout, and sending the malicious packet that causes the buffer overflow. We develop a variant of this attack where the initial phase is omitted and directly different malicious packets are sent for the memory layouts of different Apache versions.

2. Flooding Attack. This class of attacks consists of sending a flood of network requests to a server program to cause DoS.

2.1. *Ping flood*. The attack attempts to saturate a network by sending a continuous series of ICMP echo requests (pings) over a high-bandwidth connection to a target host on a lower-bandwidth connection to cause it to send back an ICMP echo reply for each request. The variants of this attack type use different packet sizes (64, 1024, 4096, 16000, 65000 bytes), and different inter-packet intervals (1 ms, 10 ms, 100 ms, 1 s).

2.2. *Smurf*. The attack sends a large volume of ICMP echo (ping) traffic to the broadcast addresses of well-populated "intermediate" networks with the source IP address spoofed to match that of the intended victim host. On receiving the echo request, each host in the intermediate network responds with an echo reply to the attacked host, flooding both the host and its network. Variants of the attack use different sizes of echo packets (512 and 1024 bytes).

3. Script-based Attack. Script programs running on a web server get user inputs and then invoke shell commands, system commands, or other programs to accomplish some tasks. If the program doesn't validate the input string before it transfers the input data to shell or system commands, this class of attacks may allow remote command execution. In our experiments, we probe the vulnerability in *open()*, and *system()* functions in the Perl CGI scripts to either overwrite or delete files, or inject executable code.

5.3. Performance Evaluation

This experiment is divided into two sets – without and with attacks injected. Different configurations of the EDs are tested. For economy of space, for the attack case, only the results with the Apache chunk attack and open SSL attack are shown. Also these attacks trigger the most number of detectors and therefore provide a worst case performance measure.

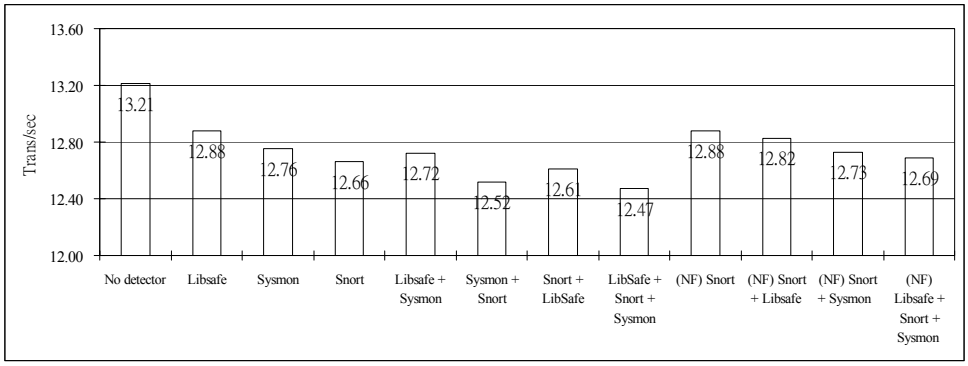


Figure 10. Performance measurement in transactions per second for electronic store front client application with different combinations of EDs

For the no attack case, 30 transactions are run concurrently. The number of transactions completing per second is measured for the 8 possible combinations of EDs. For the combinations with multiple EDs, the CIDS manager is present along with all the components mentioned in Section 4.1. These results are presented in Figure 10.

Snort as deployed with the default set of rules exhibits false alarms due to rules with Snort Rule ID (SID) 1807 and 1933. These rules are fired respectively due to the use of chunk encoded data and an inexplicable disallowing of use of the string "cart.cgi" in input. Experiments are also run after disabling these erroneous rules and are shown with the prefix "NF". The results show that the performance degradation is most significant for the active ED, Snort (4.18%) and least for the passive ED, Libsafe (2.53%). For Sysmon, the degradation is intermediate (3.46%). With the complete CIDS configuration, the performance degradation is only 5.60%, or 3.95% if the erroneous Snort rules are removed. For the rest of the paper, Snort is deployed with the full set of rules so that it satisfies the requirement of being an off-the-shelf detector.

Next we ask what the performance degradation of CIDS will be if the EDs are performing some detection. For this set of experiments, the normal workload (concurrency = 30) is run together with the attack program running continuously. The results are shown in Figure 11.

The result for the chunk attack is counter-intuitive since the performance is better for all the EDs turned on. This is because Libsafe is able to detect the chunk attack and prevent the process from core dumping. With no detector, core is dumped and the overhead of creating a large core file causes the performance degradation. With the Open SSL attack, the performance degradation is 6.33%.

Now let us try to analyze what the performance degradation is due to. In Figure 12, we show the CPU utilization by each of the components in CIDS. The total CPU utilization (user level + system level) is 100%, implying system level utilization is about 51%. The workload processes of the web server and the CGI processes have the highest utilizations. Among the CIDS components, the Matlab toolbox has the highest utilization (2.8%). This can be reduced by not invoking a separate toolbox for the Bayesian Network based Probabilistic Assurance Value computation, but performing this through native code resident within the manager itself. The Sysmon utilization is 1.5%.

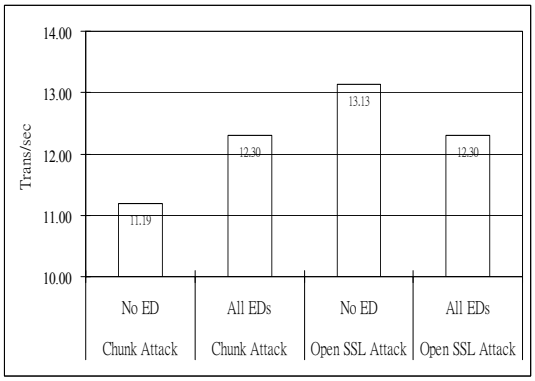


Figure 11. Performance measurements under two attack types

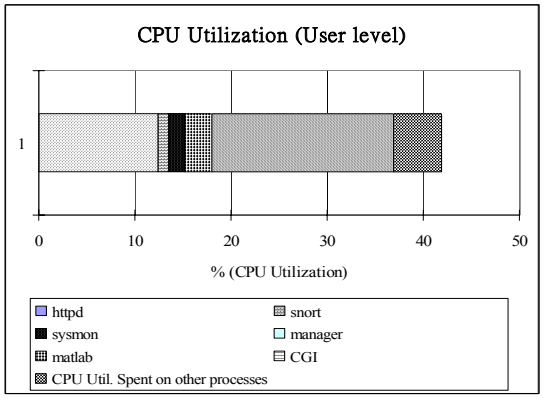


Figure 12. CPU utilization due to CIDS components and workload processes

5.4. Detection Effectiveness Evaluation

In this section, we present our experiments to evaluate the effectiveness of CIDS with respect to the false positives and

Table 1. Cases of missing alarms for different attack types in different EDs and CIDS

	Snort	Libsafe	Sysmon (Signal)	Sysmon (File)	CIDS
No attacks	Yes (1807,1933)	No	No	No	No attack
Open SSL	Yes (1881,1887)	No	Yes	*R1*	Yes
Open SSL variant	No	No	Yes	*R1*	Yes
Apache Chunk	Yes (1807, 1808, 1809)	Yes	Yes	*R1*	Yes
Smurf 1000	Yes (499)	No	No	No	Yes
Smurf 500	No	No	No	No	No
Ping Flooding	Yes (523, 1322)	No	No	No	Yes
Script	No	No	No	Yes	Yes

false negatives for the attack types and variants presented in Section 5.2.

First, we show the incidence of missing alarms with different EDs and under CIDS in Table 1. A "Yes" in a cell indicates the attack was detected. With the Snort column, the ID of the Snort rule that detected the attack is shown. Sysmon (Signal) implies the illegal signal interception function of Sysmon and Sysmon (File) implies its file access detection function. Smurf 1000 and 500 refer to ping packets of sizes 1 kB and 0.5 kB respectively. For both, the outgoing traffic from the attacker host is 114 kB/s and 10 broadcast addresses are used.

Snort throws false alarms for the normal transaction due to rules 1807 and 1933. CIDS detects 6 of the 7 variants of attacks which is better than what any individual ED can accomplish. The cells marked R1 imply that Sysmon was unable to detect a file access (creation) since the attack was not successful in reaching that step and instead crashed the Apache process. The smurf attack is undetectable by any of the EDs if it uses small enough packets, being sent at a high rate. Snort's detection relies on finding patterns in packets and doesn't detect dynamics like network flow rates. The importance of having Sysmon is borne out by the fact that the Open SSL variant and script vulnerability attacks could only by detected by it.

Next, we run experiments to detect the incidence of false alarms. We create three variants of the normal transaction: placing items in shopping cart (*cart.cgi*), clearing shopping cart (*delete.cgi*), contacting store owner (*contact.html, formmailer.cgi*). The scripts used in each are mentioned alongside. The first and third throw false alarms in Snort (rules 1933 and 884 respectively) while the second throws false alarm in Sysmon (File) since a file is being removed. CIDS shows no false alarms. It is clear that to make this result more meaningful, a much larger set of legal transactions will have to be generated and tested.

5.5. Attack Propagation Speed

The timing of the different events associated with the Open SSL buffer overflow attack with the Bayesian network based Inference Engine are shown in Figure 13. Snort Rule 1881 (SID 1881) corresponds to the triggering of the rule for the initial web server version query and SID 1887 corresponds to the rule that checks for the string "TERM=xterm" in the malicious packet. Libsafe is unable to detect this attack and as a result, the process crashes dumping core which is detected by Sysmon. It is observed that the time to launch the counter attack is higher for the Bayesian Network based Inference Engine (5.01 s against 3.97 s for the graph-based inference engine). This is due to the longer time to invoke the Bayesian Network toolbox, and the more expensive computation.

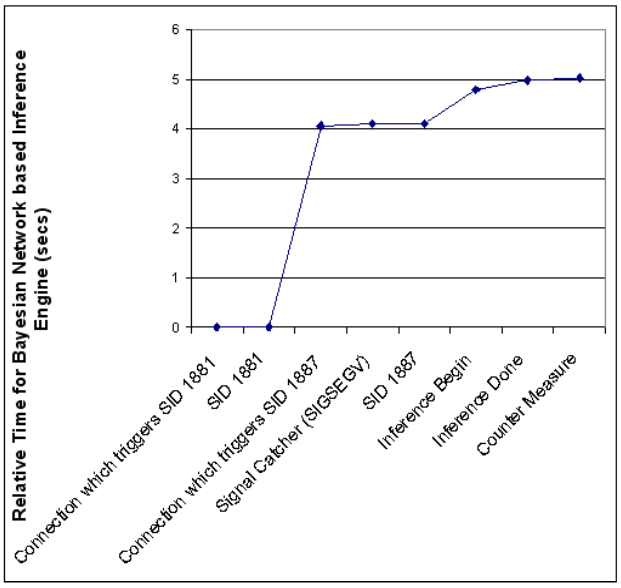


Figure 13. Timing for Open SSL attack with Bayesian Network based Inference Engine

6. Conclusions

In this paper, we have presented the architecture of a distributed system for intrusion detection called CIDS, which employs multiple elementary detectors and combination of their alerts to make an accurate determination of intrusion. Then we presented an instantiation of this architecture with three elementary detectors and a manager with a graph-based

and a Bayesian network based inference engine. We evaluated the system under a real-world web based e-commerce application and three classes of attacks. CIDS was found to bring down the incidence of missing alarms and false alarms with negligible impact on the performance.

We are currently exploring how to set up the Rule Objects for different attack classes in an automated manner. The approach uses a feedback control loop to adjust the weights or probabilities in the rules. We are developing a larger set of test cases to carry out statistically large set of experiments to measure false positives and false negatives in CIDS. We are adding a timing module to estimate the speed of propagation of attacks and augmenting the Response Engine to have a choice of responses which will be decided based on the timing information.

Acknowledgements

We would like to acknowledge the help of Eugene Spafford, Arif Ghafoor and James Joshi for several illuminating discussions on intrusion tolerance and pointers to related work. Our thanks are also due to Alan Fern for pointing us to the Bayesian Network approach to the inferencing problem.

References

[1] Curtis A. Carver, John M.D. Hill, and Udo W. Pooch, "Limiting Uncertainty in Intrusion Response," Proceedings of the 2001 IEEE Workshop on Information Assurance and Security United States Military Academy, West Point, NY, 5-6 June, 2001.

[2] "Apache Chunk Buffer Overflow Attack". At: http://httpd.apache.org/info/security_bulletin_20020617.txt

[3] F. Cuppens and R. Ortalo, "LAMBDA: A Language to Model a Database for Detection of Attacks", In Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000), Toulouse, France, October 2000.

[4] F. Cuppens and A. Miege, "Alert Correlation in a Cooperative Intrusion Detection Framework", In IEEE Symposium on Security and Privacy, Oakland, USA, 2002.

[5] M. Dacier and Y. Deswarte, "The Privilege Graph: an Extension to the Typed Access Matrix Model", in European Symposium in Computer Security (ESORICS'94), (D. Gollman, Ed.), Lecture Notes in Computer Science, 875, pp.319-334, Springer-Verlag, Brighton, UK, November 1994.

[6] M. Dacier, Y. Deswarte and M. Kaâniche, "Models and Tools for Quantitative Assessment of Operational Security", in 12th International Information Security Conference (IFIP/SEC'96), (S.K. Katsikas and D. Gritzalis, Eds.), pp.177-186, Chapman & Hall, Samos (Greece), May 1996.

[7] O. Dain and R. Cunningham, "Fusing a Heterogeneous Alert Stream into Scenarios", In Proc. of the 2001 ACM Workshop on Data Mining for Security Applications, pages 1-13, Nov. 2001.

[8] T. Verwoerd and R. Hunt, "Intrusion Detection Techniques and Approaches", In Computer Communications vol. 25, Issue 15, 2002.

[9] "Avaya Labs Research - Projects: Libsafe", At http://www.research.avayalabs.com/project/libsafe

[10] "SourceForge.net: Project Info - The libpcap project", At: http://sourceforge.net/projects/libpcap

[11] Kevin Murphy, "Bayes Net Toolbox for Matlab", At: http://www.ai.mit.edu/~murphyk/Software/BNT/bnt.html

[12] Kevin Murphy, "Tutorial on Bayesian Network Toolbox", At: http://www.ai.mit.edu/~murphyk/Software/BNT/BNT_mathworks.ppt

[13] Peng Ning, Yun Cui, Douglas S. Reeves, "Constructing Attack Scenarios through Correlation of Intrusion Alerts", In Proceedings of the 9th ACM Conference on Computer & Communications Security (CCS 2002), pages 245-254, Washington D.C., November 2002.

[14] Rodolphe Ortalo and Yves Deswarte and Mohamed Kaaniche, "Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security", Journal of Software Engineering, vol. 25, no. 5, pp. 633-650, 1999.

[15] M. Roesch, "Snort – Lightweight Intrusion Detection for Networks", In Proceedings of USENIX LISA'99, November 1999.

[16] "Apache OpenSSL Attack". At: http://www.cert.org/advisories/CA-2002-27.html

[17] S. Staniford, J.A. Hoagland and J.M. McAlerney, "Practical Automated Detection of Stealthy Portscans", In the Journal of Computer Security, Volume 10, Issues 1/2, 2002, pp. 105-136.

[18] S. Templeton and K. Levit, "A requires/provides model for computer attacks", In Proc. of New Security Paradigms Workshop, pages 31-38, September 2000.

[19] A. Valdes and K. Skinner, "Probabilistic alert correlation", In Proc. of the 4th Int'l Symposium on Recent Advances in Intrusion Detection (RAID 2001), pages 54-68, 2001.