ABSTRACT

THOMAS, ASHLEY. Adaptive Real Time Intrusion Detection Systems. (Under the direction of Dr. Wenke Lee.)

A real-time intrusion detection system (IDS) has several performance objectives: good detection coverage, economy in resource usage, resilience to stress, and resistance to attacks upon itself. In this thesis, we argue that these objectives are trade-offs that must be considered not only in IDS design and implementation, but also in deployment and in an *adaptive* manner. A real-time IDS should perform performance adaptation by optimizing its configuration at run-time. We use classical optimization techniques for determining an optimal configuration. We describe an IDS architecture with multiple dynamically configured front-end and back-end detection modules and a monitor. The front-end does the real-time analysis and detection and the less time-critical tasks may be executed at the backend. In order to do performance adaptation, the front-end is modified to have two modules: performance monitoring and dynamic reconfiguration. The IDS run-time performance is measured periodically, and detection strategies and workload are dynamically reconfigured among the detection modules according to resource constraints and cost-benefit analysis. The back-end also performs scenario (or trend) analysis to recognize on-going attack sequences, so that the predictions of the likely *forthcoming* attacks can be used to pro-actively and optimally configure the IDS.

The adaptive IDS results showed better performance when the operating conditions changed and the IDS was stressed or overloaded. By reconfiguring, the adaptive IDS minimized packet drops and gave priority for critical attacks, with relatively higher damage cost, thereby ensuring maximum value for the IDS. The overheads involved for monitoring as well as reconfiguration was found to be negligible.

Keywords: real-time intrusion detection, performance metrics, performance adaptation, optimization.

ADAPTIVE REAL TIME INTRUSION DETECTION SYSTEMS

by

ASHLEY THOMAS

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Networking

Raleigh

2002

APPROVED BY:

Chair of Advisory Committee

To Amma and Daddy.

BIOGRAPHY

Ashley Thomas completed his Bachelor Degree in Electronics and Communications Engineering in 1997 from the University of Calicut, India. Presently, he is pursuing a Master of Science degree in Computer Networking at the North Carolina State University, Raleigh.

ACKNOWLEDGEMENTS

My sincerest thanks to Dr. Wenke Lee for his guidance all through my research work. It has been a pleasure working under his guidance. I also thank Dr. Douglas Reeves and Dr. Peng Ning for agreeing to be on my thesis research committee and providing valuable input.

I would also like to thank my friends Bobby George, Sunny, Niranjan, Vinay Mahadik, Somani Ram for their help and support.

Finally, I would like to thank my family - Amma, Daddy, Milanchechi, Jewel, Elsamma aunty, Philomma aunty, Mariette aunty, Anniemma aunty, Cama aunty and the IMS group for their prayer and support which was and continues to be a tremendous source of strength.

Thank you.

Contents

List of Figures

Chapter 1

Introduction

The ever increasing threat from hackers, competitors and other enemies has made the security of computers and computer networks, a prime concern for any organization. The main objectives of security is to protect the integrity, confidentiality and availability of resources of the organization. The resources may be network bandwidth, computers, or confidential corporate data. The most effective way to ensure security is by the layered approach; also known as Defense-in-depth. This approach has basically three elements;

- Prevention.
- Detection.
- Response.

Prevention is usually done using perimeter control mechanisms like Firewalls or Router access control lists (ACLs). Although perimeter control is the most important element of the Defense-in-depth model, they cannot prevent 100 percent of the attacks. This maybe due to human errors like misconfigurations of firewall or attacking through an unfiltered port. It is the function of the Detection element to detect when the attacker is successful in getting past the perimeter control. This is analogous to having burglar alarms in our houses. Whether the house is locked, or is unlocked due to oversight, the burglar alarm notifies the owner when there is any trespassing. The third element is concerned with how the detection element reacts to an intrusion.

1.1 Motivation

An IDS is a "mission-critical" system that needs to be effective and available. More specifically, its *performance objectives* include: real-time detection and notification, good detection coverage, economy in resource usage, and resilience to stress [23]. Since sophisticated adversaries may try to first evade or even subvert IDSs when launching their intended attacks, another important performance objective is that an IDS must resist attacks upon itself [22, 20]. These objectives can be conflicting goals. For example, for broad coverage and high detection accuracy, an IDS needs to perform stateful analysis on a lot of audit data and the number of rules or signatures also need to be large. This requires a large amount of resources (in both memory and detection time). A resource-intensive IDS is then vulnerable to stress and overload attacks. We therefore need to carefully consider the trade-offs.

Current trend lays emphasis on fine tuning the IDS configuration to suit the environment and operating conditions where the IDS will be deployed for better performance. Some IDSs are carefully designed to be very "light weight" or are specially configured with high-end hardware (e.g., RealSecure with AppSwitch [29]) to cope with high-speed and high-volume traffic. However, as the results of the experiments in later sections show, as long as an IDS is statically configured, it can be overloaded. If the overloaded situation persists it might lead to the IDS dropping packets and consequently missing the attacks. Such overloading scenarios may be completely unintentional, being the effect of fluctuations in traffic load on the network. On the contrary an intelligent adversary can deliberately overload the IDS to a point that it will miss the intended attack with high probability, in order to avoid detection.

Overview of the solution

We advocate enabling an IDS to provide *performance adaptation*, that is, the best possible performance for the given operation environment. It is extremely difficult, if not impossible, for an IDS to be 100% accurate [2]. The optimal performance of an IDS should be determined by not only its ROC (Receiver Operating Characteristics) curve of detection rate versus false alarm rate, but also its cost metrics (e.g., damage cost of intrusion) and the probability of intrusion [7]. Accordingly, performance adaptation means that an IDS should always maximize its cost-benefits for the given (current) operational conditions. For

example, if an IDS is not able to keep up with the traffic on the network and therefore is forced to miss some intrusions (that can otherwise be detected using its "signature base"), due to stress or overload attacks, it should still ensure that the best value (or minimum damage) is provided according to cost-analysis on the circumstances. As a simple example, if we regard buffer-overflow as more damaging than port-scan (and for the sake of argument, all other factors, for example., attack probability, detection probability, are equal), then missing a port-scan is better than missing a buffer-overflow. The cost factors of various intrusions are calculated according to site-specific security policies and priorities [7, 13].

The performance adaptation is achieved by continuous run-time performance measurement and monitoring, and dynamic reconfiguration mechanisms. This ensures that detection rules and analysis tasks of higher priority gets preference if the IDS is unable to do all the analysis and detection. A knapsack optimization algorithm is employed to select the most appropriate configuration, i.e. having maximum value, under the current constraints. The performance adaptation is implemented on two IDSs: Bro [20] and Snort [25]; and the results are compared against the original non adaptive version. These IDSs are chosen because they represent the state of the art real-time NIDS technology and are open source. The experiments are conducted using LARIAT traffic generation testbed.

Contribution to the field

Our research work is one of the pioneering work in the field of performance adaptation in Intrusion detection. We list some of the contributions:

- Prototype for a real-time adaptive IDS is provided derived from two different IDSs-Bro, which is a stateful IDS and is event based; and Snort, which does packet based analysis.
- Optimization algorithms are implemented for reconfiguration, to get the best possible IDS configuration at run-time and the results are described.
- Results are provided to show how the prototype adaptive IDSs perform in different overload scenarios compared to the traditional non adaptive ones.
- We suggest the various parameters or metrics that an IDS should maintain for performance monitoring. Also, a model for a Real-time adaptive IDS is provided.

- From our experience with tailoring Bro and Snort, we suggest some features a model performance adaptive IDS should have.

1.2 Summary of remaining chapters

Chapter 2 gives an overview of IDSs, their classification, their performance constraints or bottlenecks and various attacks on IDSs. Chapter 3 states the research problem and discusses the importance of it. Chapter 4 deals with analysis of IDS performance issues, objectives, and the need for performance adaptation. Chapter 5 describes the experiments done on statically configured IDS and explains the results. Chapter 6 describes the architecture for performance adaptation and also the various components. Chapter 7 discusses how to enable performance adaptation in real-time IDS and describes the prototype real-time adaptive IDSs. Chapter 8 has the results of the experiments performed on the developed prototypes. It also lists some difficulties we faced while building the prototype adaptive IDS and also shows a wish list for a model adaptive IDS and Chapter 9 concludes the thesis with a summary.

Chapter 2

Background

This chapter gives a background on Intrusion detection systems (IDS) and the various classifications of IDSs. Real time ID systems and their performance bottlenecks are explored. It also gives an overview about the various attacks targeting the Real time IDS. Finally, a discussion about general IDS configuration is included.

2.1 Intrusion Detection Systems

An Intrusion detection system is a Detection element and is an important part of the Defense-in-depth strategy. Intrusion detection can be defined as: [1]

Intrusion detection is the process of identifying and responding to malicious activity targeted at computing and networking resources.

The definition briefly lists the functions of an IDS; i.e.

- Identifying the malicious activity and
- Responding to it.

The means of identifying "malicious" activity differs from one IDS to another and it can be done while the activity is going on or after it has occurred. The response that an IDS takes also varies from reporting or logging into some database to making counter attacks.

2.2 Classification of Intrusion detection systems

An IDS can be classified in various ways [9]; i.e. based on detection method, detection timing, type of response and type of audit data used.

Based on the type of detection method used by the IDS, it can be classified into the following types:

- Anomaly based IDS: This class of IDSs detects malicious activity on a host or on the network by looking for large deviations from normal or acceptable behavior.
- Misuse based IDS: This class of IDSs detects malicious activity by monitoring the audit data for misuse signatures or known attack patterns.
- Protocol based IDS: This is a class of IDS under the Network based IDS category. This class of IDS detects malicious activity by doing analysis based on protocols.

Another classification of an IDS is based on the detection timing. The different types are:

- Real time IDS: The real time IDS detects the attacks in real time, i.e when the attack is unfolding, and reports them or takes appropriate action. The real time IDSs operate under time constraints. The time available for analyzing an event is bounded.
- Offline IDS: The offline IDS operates on log files consisting of events. It does not have time constraints and therefore can do a more thorough and detailed analysis but the downside is that damage is already done.

The response of an IDS in the event of detecting a malicious activity also varies. Accordingly they can be classified as:

- Passive IDS: Traditionally, IDS operates in a passive mode. In the event of detecting an attack it just raises an alarm and logs the details.
- Active IDS: On detecting an attack, these IDSs take different actions which are specified. The actions can vary from bringing a misbehaving TCP connection down by sending RST packets to either endpoint or interacting with the firewall to block certain IP addresses or port.

Based on the type of audit data that it monitors, an IDS can be classified as:

- Host based IDS: A host based IDS monitors the host system events like what files were accessed, what applications were executed, or the pattern of system calls made by certain application.

- Network based IDS: A network based IDS monitors the network activities by capturing and analyzing audit data (e.g., `BSM` [28] or `libpcap` [15] stream) to determine whether there is an attack occurring.

- Network Node based IDS: This class of IDS resides on main network nodes and analyze only the traffic directed to the node. These type of IDSs are implemented in the TCP/IP stack of the host Operating system and are also called Stack-based IDS.

2.3 Real-time NIDS

A Network based IDS or NIDS operates on the raw packets seen on the network that it is monitoring. As opposed to offline NIDS, a Real-time NIDS does the detection in real-time; i.e. when the attack is occurring. This is really helpful especially when the attack has multiple stages; detecting it at an early stage can help to prevent it from causing damage. A NIDS is usually placed at strategic points on the network, for e.g. in the Demilitarized zone (DMZ), where it can see all the traffic bound to the network. These IDSs work in promiscuous mode, which enables them to capture all the traffic irrespective of the destination address of the packet. The NIDS is like a surveillance tower, monitoring all the nodes in a network. In order to analyze how a certain host will behave on receiving a packet, the NIDS needs to do a minimum amount of processing like IP fragmentation reassembly, TCP stream reassembly and keeping state of different active connections.

When NIDS operates as a real-time system, there are some additional design requirements. While not compromising on the accuracy, the IDS functions have to be done under time deadlines and with limited resources to provide a certain accepted level of performance guarantee.

2.3.1 Constraints and Bottlenecks in Real-time NIDS

Like any other real-time system, a real-time IDS has many resource constraints [16]. Some of the important resources of a real time IDS are CPU speed, memory size, memory bus bandwidth, bus bandwidth for the network interfaces, long term storage etc.

Since the NIDS processes all the packets on the network, as the traffic volume increases, the time available for packet processing becomes lesser.

Some of the common weak points or choke points of a Real time IDS are [16]:

- Alert or Alarm channel: One of the main responses of any IDS is to log to a non volatile storage like disk. This is a time consuming operation and a high frequency of alerts can act as a bottleneck.
- Critical Path: For every IDS there is an analysis path which takes much more than the average CPU cycles. This path maybe due to detailed analysis tasks like IP fragmentation reassembly or TCP reassembly, or due to a large number of signatures/rules for a particular type of packet, or due to a computationally expensive operation like pattern matching in snort.
- Stressing the database: For IDSs that do stateful analysis, as the number of concurrent connections increases, the database accesses will take more time thus overloading the IDS.
- Packet capture mechanism: As the network traffic goes higher and higher, the packet capture mechanism can become a bottleneck.

2.3.2 Attacks on the IDS

While Intrusion detection systems are deployed to detect attacks on the network, they themselves can be attacked. Some of the attacks that can be done on an IDS are overload, crash or Evasion attacks. [22, 20, 27].

- Overload attacks: The strategy of the attacker in this type of attack is to identify a weak spot or critical path in the IDS and overload it. The detection performance cannot be guaranteed when the IDS is in stress and there is a non negligible probability that the attacks during that time will go unnoticed.
- Crash attacks: These are the attacks in which the attacker tries to crash the IDS out of operation. This maybe done by finding some loophole or bug after careful study of the IDS code or functionality, especially with open source IDSs.
- IDS Evasion: Various IDS evasion techniques exist that takes advantage of the fact that an NIDS does not have a topology knowledge of the network it is watching or

that certain IDS do not do IP fragmentation reassembly or TCP reassembly properly. While crash attack and overload attacks are aimed at the IDS, similar to a DoS, IDS Evasion is to get past the IDS, unnoticed.

Researchers are developing attack resistance techniques. Many evasion attempts can be foiled if an IDS uses stateful analysis and employs a network traffic *normalizer* [10].

2.4 IDS Configuration

A set of parameters, whose values determine the an IDS performs constitute the "Configuration" of an IDS. The values of these parameters at any instance of time determines the performance of an IDS, as well as its value. Some of the common IDS parameters are:

- Set of detection rules: It is very critical that the rule set that one uses has to be updated and tuned for the environment. Detection rules vary in complexity, the criticality of the attack they detect etc.
- Analysis modules or tasks: Analysis modules in the form of plug-ins are common with IDSs. The user specifies in the configuration file the ones to be loaded.
- Memory limits. These parameters sets the different memory limits an IDS should be using for different modules like TCP stream reassembly, IP fragmentation reassembly etc.
- Packet capture filter: This is a parameter that is specified to the packet capture mechanism to convey what packet the IDS is interested in. By eliminating the unwanted packets at the earliest, the performance can be improved a lot.
- Different timer values: There are different set of timers associated with an IDS. Each of them indirectly affects the performance of the IDS and also the amount of memory the IDS needs etc. For example, the fragmentation timer of an IDS tells an IDS about the duration it should keep an incomplete set of fragments before all of them arrives. Ideally, this timer value should be exactly same as what the target hosts would be having. But since the IDS is in a hot spot, it is trade off to be made what the timer value needs to be.

It is a very critical factor for an IDS to have an optimal configuration. There is a tendency among users to configure the IDS and get it running with the default configuration that comes with the IDS distribution. This usually leads to non optimal performance of the IDS and thereby reducing its value.

The next chapter states the problem we are interested in and some literature reviews. The importance of the problem is discussed as well.

Chapter 3

Research Problem Statement and State of the art

This chapter briefly states the research problem this thesis addresses. The importance of the problem is discussed. Related work and why they fail to address the issue completely, is discussed.

3.1 Problem and its importance

Current real-time IDSs are statically configured and do not have the features to adapt their configurations and work load at run-time, according to the changing operational conditions. Therefore, as the operating conditions change due to fluctuation in the traffic profile or when under an overload attack, the original configuration becomes a non optimal one and reduces the value of the IDS.

This problem is most often dismissed with the argument that it is okay if the IDS drops some packets when there is a traffic spike. But the fact is that the attacker can deliberately increase the traffic to "hide" the attack. In such cases the reason of deploying the IDS itself is lost. The report from NSS group [9] shows that most of the current IDSs are not capable of keeping up with high loads or moderate loads in some cases. The above points suggests the need for an IDS to react to the changes in operating conditions and adjust itself to give maximum benefit.

Under overloaded circumstances, IDSs do not provide guarantee whether an attack traffic will be detected or not. Also, due to the site specific policies and rule set, the attacker

might not exactly know the traffic load needed to overload the IDS in order that his attack may go undetected. Paxson [20] mentions that this uncertainty itself can be a defense mechanism against the attacker. However, that is not a foolproof method and do not guarantee detection of critical attacks. The uncertainty of the attack going undetected is same for a critical attack as for a non critical one. Besides the attacker can make some intelligent guessing regarding the traffic load needed or just go for a very high traffic flooding.

The other approaches mentioned in the previous chapter do not completely solve the above problem. Although high end hardware platforms can be used for data capturing to ensure "no packet filter drops" [20], an IDS, with the often site-specific ID logics implemented as application-level software, can still be overloaded due to high volume of events or large number of nuisance alarms. A few approaches are discussed below:

There are different approaches for tackling the overwhelming of IDS due to high volume of traffic. Such overwhelming can be caused due to high spikes in traffic or due to intentional overloading.

- Partitioning the traffic: Kruegel et al. proposed this approach for monitoring a high speed network link [12]. Compared with simple load-balancing, this approach is to partition the traffic meaningfully to a distributed set of sensors each equipped with a set of detection rules. This approach is well suited for monitoring high speed traffic links given the traffic distribution is more or less same. Even with this distributed approach it is noted that the system is vulnerable if the configuration for splitting is static, resulting in some of the sensors being overloaded unless some dynamic adaptation is incorporated.

- Load shedding: Paxson suggested that load-shedding may help a real-time IDS defend against overload attacks [20]. For e.g. the IDS could stop analyzing HTTP to reduce load. Load shedding definitely reduces the load of the IDS in stressed situations and packet drops can be avoided. But when such load shedding is static it turns out to be non-optimal. For example, when traffic load increases if the IDS always sheds HTTP or any other protocol, the IDS will not be utilized well. Besides the attackers can take advantage of the resulting "blindness" of the IDS.

- Distributed approach: Several enterprise-wide and Internet-wide distributed IDSs [21, 30] and agent-based architectures [4, 8] have been proposed to address the issues

of detection coverage and workload distribution. For example, in EMERALD [21], ID modules are deployed and configured in a hierarchical fashion according to the enterprise network topology.

Instead of placing a single IDS at a critical point where all the traffic can be seen, say Demilitarized zone, this approach suggests placing a number of IDSs at different strategic locations through out the network to do the monitoring. In this approach there is additional problem of doing correlation of data seen at different parts of the network and making a detection, especially when the attack is distributed. Besides, this approach is still vulnerable to the intentional overload attack by a strategic attacker. This happens when the flooding traffic is just specific to a subnet rather than to different subnets.

Our research attempts compliments these research efforts because performance monitoring and performance adaptation via dynamic reconfiguration are the necessary techniques for an IDS to adaptively resist attacks. It is often more appropriate to evaluate an IDS using the damages (costs) it has prevented [7, 13]. We use cost-benefit analysis to determine the best IDS configurations given the resource constraints.

Chapter 4

IDS Performance Analysis

In this chapter, a model for the real time IDS (RTIDS) is discussed and its performance trade offs are analyzed from an optimization and control perspective as described in [5]. Statically configured IDS behavior is compared with Dynamically configurable IDS behavior. Finally, the results of experiments of IDSs (Snort and Bro) having static configurations with an overload attack is explained.

The RTIDS can be seen as a queuing system with one or more servers serving one or more queues. We analyze the single server single queue model since that models all the present real time IDSs. The queue is the packet capture buffer; for e.g. the buffer associated with the libpcap mechanism. The individual packets wait in the buffer until it is processed by the server; i.e. the IDS. The following section explains some definitions needed throughout the chapter.

4.1 Definitions and Preliminaries

Audit Records Audit records (or audit events, e.g., packets) are categorized according to their types. Examples of (high level) types are `tcp connection attempts`, `tcp connection established`, `http requests`, `icmp echo request`, etc. There are a total of N record types that an IDS accepts. Each audit record is either part of a normal session, or an attack of a certain label. We denote E_i as an arbitrary audit record of type i. Audit record types are characterized by their prior probabilities π_i, which denote the probability that a given record belongs to type i.

Attacks There are a certain number of attacks associated with each audit event type. For example, a `tcp connection attempt` can be part of a "port-scan" or "syn flood"; or an `icmp echo request` can be part of a "ping scan" or "ping flood" or a "ping of death". Denote N_i as the number of "known" attacks associated with audit event type i. That is, for type i, the IDS has analysis tasks and detection rules for only N_i attacks (other attacks are "unknown" to the IDS). We denote the attacks as A_{ij}, where $j = 1, 2, \cdots, N_i$. We say that $E_i \leftarrow A_{ij}$ when A_{ij} is present in E_i, and $E_i \leftarrow A_{i0}$ when audit event E_i is normal. There is a total of $\sum_{i=1}^{N} N_i$ known attacks to the IDS. Attacks are characterized by the following quantities:

- Prior Probability: The probability p_{ij} that an event of type i contains A_{ij}, that is, $p_{ij} = (E_i \leftarrow A_{ij})$. Clearly, from the perspective of IDS, $\sum_{j=0}^{N_i} p_{ij} = 1, \quad i = 1, 2, \cdots, N, \ j = 1, 2, \cdots, N_i$, where p_{i0} is the prior probability that an audit record of type i is normal, that is, $p_{i0} = (E_i \leftarrow A_{i0})$. This value is not static and is updated depending on the traffic and attack history. The initial value of this parameter can be calculated based on some data set from the same network and is continuously updated at run-time. The calculation of the initial value may be not very accurate (depending on the data set) but more accurate values are obtained when it is continuously calculated at run-time.

- Damage Cost: The cost associated with attack A_{ij} being missed by the IDS, denoted as $\mathcal{C}_{ij}^{\beta}$. This value is decided according to site policies and damage level of the attack. These values are static and do not change during run-time. It is a relative damage cost with respect to the other attacks. These are calculated while doing the risk analysis for the organization.

- False Alarm Cost: An alert detected by the IDS is called a False alarm, when a non-intrusive event is classified as intrusive. The cost associated with a response triggered by a false alarm that attack A_{ij} is present, denoted as $\mathcal{C}_{ij}^{\alpha}$. This value is site dependant and does not change with time, i.e. it is static. This value consists of the labor cost involved when a false alarm is raised, as well as the cost involved with active responses, like blocking a valid ip address.

Analysis Tasks Each audit record is subject to a number of analysis tasks in the IDS, including data (pre)processing, rule checking (i.e.,intrusion detection), and logging. Denote

K_i be the (maximum) number of tasks for audit event type i. The tasks are denoted as R_{ij}, where $j = 1, 2, \cdots, K_i$. The notation, $R_{ij} \stackrel{r}{\leftarrow} A_{ij}$ means that a Detection Rule R_{ij} reports the presence of attack A_{ij} in audit event E_i. Similarly the notation $R_{ij} \stackrel{r}{\leftarrow} A_{i0}$ means that R_{ij} reports event E_i to be normal. The detection rules are characterized by the following quantities:

- The False Alarm Rate of R_{ij} denoted by α_{ij} is defined as $\alpha_{ij} = (R_{ij} \stackrel{r}{\leftarrow} A_{ij} \mid E_i \leftarrow A_{i0})$
- The False Negative Rate of R_{ij} denoted by β_{ij} is defined as $\beta_{ij} = (R_{ij} \stackrel{r}{\leftarrow} A_{i0} \mid E_i \leftarrow A_{ij})$

Each task R_{ij} (regardless whether it is a detection rule or not) is also characterized by its Computation Time t_{ij}.

System Configuration The run-time configuration of an IDS is characterized by the collection (union) of its analysis tasks. That is, IDS configuration $\mathcal{P} = \bigcup_{i=1,\cdots,N} \mathcal{P}_i$, where $\mathcal{P}_i$ is the collection of tasks for event type i, that is, $\mathcal{P}_i = \bigcup_{j=1,\cdots,K_i} R_{ij}$ (note that not all tasks are detection rules). A *statically configured* IDS has a fixed set of tasks regardless of changes in run-time conditions.

4.2 Performance Metrics

Expected Value The purpose of a real-time IDS is to detect intrusions and prevent damages. Besides its statistical accuracy, as shown by the ROC curve, an IDS should be evaluated according to its value (or cost-benefit). The value of an IDS is the cumulative sum of individual values of all the detection rules For each attack A_{ij}, an IDS equipped with the detection rule R_{ij} (and the necessary preprocessing and logging tasks) for A_{ij} provides the expected value:

$$\mathcal{V}_{ij} \quad = \quad \mathcal{C}_{ij}^{\beta} \pi_i p_{ij} (1 - \beta_{ij}) - \mathcal{C}_{ij}^{\alpha} \pi_i (1 - p_{ij}) \alpha_{ij} \tag{4.1}$$

The first term is the loss (damage) prevented because of true detection, and the second term is the loss incurred because of false alarms. The total value of an IDS depends on its configuration, that is, its collection of analysis tasks and hence the attacks that it "covers". For the "default" configuration $\mathcal{P}$ that covers all known attacks, the value is $\mathcal{V}(\mathcal{P}) = \sum_{i=1}^{N} \sum_{j=1}^{N_i} \mathcal{V}_{ij}$.

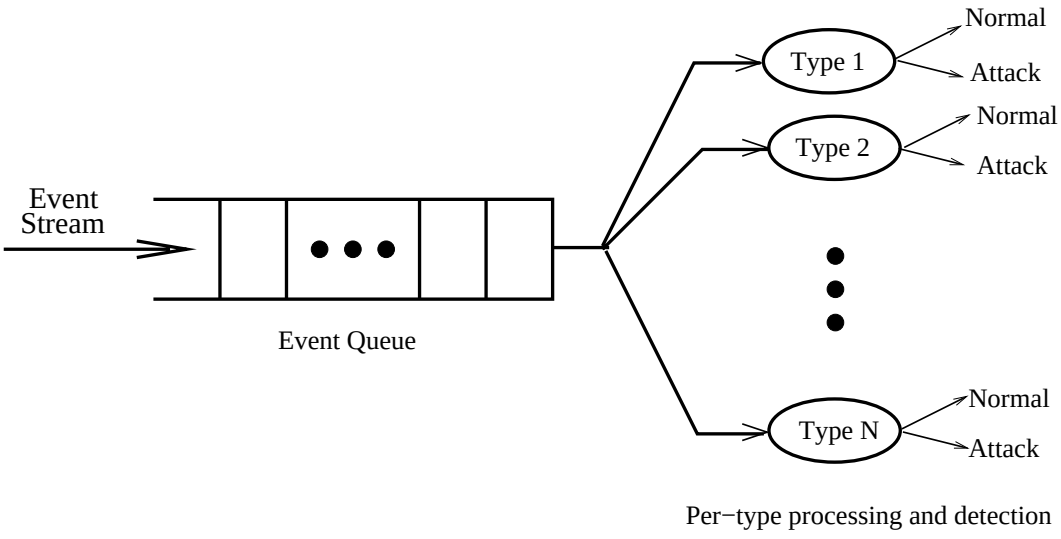

Figure 4.1: The IDS Processing Flow. All events are directed to a common queue, but the nature of the service performed on each event depends on event type.

Response Time Figure 4.1 shows a generic IDS processing flow. Upon arrival in the system, audit records are placed in a (common) queue (e.g., the `libpcap` buffer). The queue has only one server, the audit data processing and intrusion analysis unit. The nature of the service performed on an audit record item depends on its type; i.e. an icmp packet will be following a different path of processing than a tcp or udp packet. That is, records of type i are only subject to the tasks belonging to $\mathcal{P}_i$. The processing and analysis tasks for each audit record are applied sequentially, as depicted in Figure 4.2. That is, each event goes through a sequence of analysis tasks. The processing of that event terminates if a detection rule R_{ij} determines that the event is (part of) an intrusion. Or the process ends when all analysis is done and the event is deemed normal. The expected system time (queuing time plus service time - [11]) for an audit record of type i' that arrives in an IDS with configuration $\mathcal{P}$ at a time when there are m_i records of type i, $i = 1, 2, \cdots, N$ in the queue is given by:

$$T(\mathcal{P}) = (\sum_{i=1}^{N} m_i T_i) + T_{i'} \tag{4.2}$$

where T_i denotes the expected service time for a record of type i. The first term, $\sum_{i=1}^{N} m_i T_i$, is the queuing time that the audit record i' sees; and the second term,$T_{i'}$, is its service time. The system time corresponds to the time interval elapsed between an audit record entering

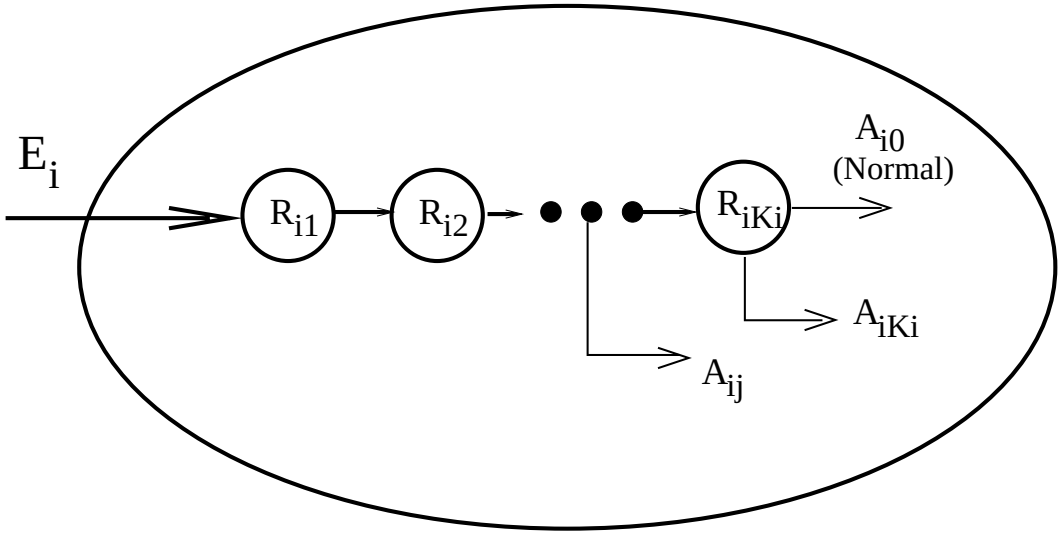

Figure 4.2: Processing of events of type i. That tasks include preprocessing, rule-checking, and logging. They are applied sequentially.

the system and a decision being made about the presence or absence of an attack in the event. That time is the response time of the IDS. In the case when the queue is close to full, $\sum_{i=1}^{N} m_i T_i \gg T_{i'}$, that is, the queuing time dominates the service time for a typical event. Equation (4.2) becomes:

$$T(\mathcal{P}) = \sum_{i=1}^{N} m_i T_i, \quad \text{where} \;\; T_i = \sum_{j=0}^{N_i} p_{ij} T_{ij}, \quad \text{and} \;\; T_{ij} = \sum_{\ell=1}^{j} t'_{i\ell} \tag{4.3}$$

T_{ij} denotes the service time of an event of type i which is matched by detection rule R_{ij}. Note that the time of common preprocessing and logging tasks for event type i is included ("factored in") into the time of the N_i detection rules ($N_i \leq K_i$), one for each known attack associated with event type i. T_{ij} is computed as a sum of time of all previous tasks because $R_{i1}, R_{i2}, \cdots, R_{iN_i}$ are applied sequentially. Each t'_{il} is the sum of the time of detection rule R_{il} (i.e., t_{il}), and the time of common preprocessing and logging tasks for event type i. Recalling that $\sum_{j=0}^{N_i} p_{ij} = 1$, we have:

$$\begin{aligned} T(\mathcal{P}) &= \sum_{i=1}^{N} \sum_{j=1}^{N_i} u_{ij} t'_{ij}, \\ \text{where} \;\; q_{ij} &= 1 - \sum_{\ell=1}^{j-1} p_{i\ell}, \;\text{for}\; j \geq 1, \quad q_{i1} = 1, \;\; \text{and} \;\; u_{ij} = m_i q_{ij} \end{aligned} \tag{4.4}$$

4.3 Performance Optimization

An IDS should provide broad detection coverage to maximize its $\mathcal{V}(\mathcal{P})$. This requires that the IDS perform a thorough analysis (e.g., do stateful packet re-assembly and analysis), and include many detection rules. This in turn requires a configuration $\mathcal{P}$ with many complex tasks, resulting in a large $T(\mathcal{P})$.

On the other hand, the main constraint in real-time intrusion detection is that $T(\mathcal{P})$ needs to be bounded. As audit events stream into the system queue (see Figure 4.1), they need to be serviced (taken off) at a rate faster than the arrival rate. Otherwise, the queue (with limited size) will be filled up, and remain so, with the not yet serviced events, thus the newly arriving events cannot be placed in the queue. This phenomenon is referred to as audit data "dropping". The consequence is that the false negative rate(s) β_{ij} of some detection rule(s) R_{ij} will increase due to missing information (evidence). When the queue is full, the IDS do not have control over which audit record is dropped; therefore false negative rates of critical attacks (having higher cost) will become high. The IDS value $\mathcal{V}_{ij}$ (see Equation (4.1)) and hence $\mathcal{V}(\mathcal{P})$ will then also decrease. Therefore, in order to provide the expected value, an IDS configuration should satisfy the constraint $T(\mathcal{P}) \leq D_{max}$, where D_{max} is the mean audit event inter-arrival time.

The goal is then to configure an IDS to provide the best value while operating under the above constraints. That is, if an IDS cannot accommodate all desirable analysis tasks (without violating the constraints), it should just include the more valuable tasks (we also assume that additional and orthogonal optimization techniques, such as rule-set ordering, can be used). For example, an IDS should always detect "buffer-overflow" and only analyze "slow scan" when time permits. More formally, the problem is to solve the following equation:

$$\max_{x_{ij}} \mathcal{V}(\mathcal{P}) = \sum_{i=1}^{N} \sum_{j=1}^{N_i} \mathcal{V}_{ij} x_{ij}$$

$$\text{subject to } \; T(\mathcal{P}) = \sum_{i=1}^{N} \sum_{j=1}^{N_i} u_{ij} t'_{ij} x_{ij} \leq D_{\max}$$

$$\text{where } \; x_{ij} = 1 \; \text{ if } R_{ij} \text{ is active in } \mathcal{P} \text{ and } \; x_{ij} = 0 \; \text{ otherwise} \tag{4.5}$$

The solution is the set of x_{ij} values, which specifies an IDS configuration by indicating which tasks should be included (active). This is known as the Knapsack problem (e.g., [14, 18]) in the optimization literature. Note that preprocessing and logging tasks in

$\mathcal{P}$ are "factored in" (included in) the detection rules in the following ways: as long as a detection R_{ij} for event type i is active, the common preprocessing and logging tasks for event type i are also included in $\mathcal{P}$; otherwise, when all detection rules for event type i are deactivated, these non-detection tasks for i can also be excluded from $\mathcal{P}$ (Alternatively, one may still want to at least log type i events after some minimum amount of processing. For simplicity sake, the time and value in such a situation is omitted).

Instead of requiring exact measurements of the parameter values (e.g., p_{ij}) when solving Problem (4.5), a more meaningful approach is to allow a value range (with upper and lower bounds) for each parameter. For a feasible IDS configuration $\mathcal{P}$ (specified by a set of x_{ij} values), there is a range of $\mathcal{V}(\mathcal{P})$ values (while the constraint $T(\mathcal{P}) \leq D_{max}$ is always satisfied) because of the ranges of parameter values. The "worst-case" is when $\mathcal{V}(\mathcal{P})$ is the minimal. The optimization problem is then to find an IDS configuration that maximizes the minimal value. In [5], we showed that we can convert the resulting robust max-min problem into an equivalent Knapsack problem, with computational properties similar to the original problem.

4.4 Static Configuration vs. Adaptation

An IDS configuration comprises of all the analysis tasks or modules (for e.g. http analysis, tcp reassembly, ip fragmentation reassembly), the corresponding packet capture filter used, and the set of signature rule set. The current IDSs are all based on static configuration. There is no way to change the configuration once the IDS is running. The current trend is still on fine tuning the IDS configuration to suit the environment where the IDS is going to be deployed. The developers and on-site engineers often use knowledge of threat models and assumptions on operation environments to make the appropriate design and customization decisions so that the IDS provides the best performance under the constraints. However, as examples in Section 5 show, current IDSs do not have the mechanisms to continuously monitor its performance and the conditions of its run-time environment. That is, they are usually statically configured in run-time. Such systems are not optimal when run-time conditions change, and are vulnerable to attacks aimed to elude IDS.

It is noted that a solution to Problem (4.5) (i.e., the optimal IDS configuration) is only valid for a given set of parameter value ranges. Among them, π_i, m_i, and p_{ij} can

fluctuate with the operating conditions (e.g., network traffic). For example, when an IDS is under "stress" (i.e., high speed and/or large volume of audit data), m_i becomes much larger and so does $T(\mathcal{P})$. In overload or DoS attacks, an attacker first generates a lot of events (that may include "nuisance" attacks) to overload the IDS, and then launches the intended attack [22, 20], say A_{ij}. The overloaded IDS may be "dropping" audit data, missing key evidence, and hence failing to detect attack A_{ij}. Or its detection is too late (slow) to prevent the damage of A_{ij}. In either case, $\mathcal{V}(\mathcal{P})$ will likely decrease by $\mathcal{V}_{ij}$.

In these "stress" and "overload attack" situations, while it is unavoidable that IDS performance will suffer (i.e., the intended (original) $\mathcal{V}(\mathcal{P})$ cannot be guaranteed), it is desirable that a new optimal $\mathcal{V}(\mathcal{P})$ (i.e., the best value possible under the new operating conditions) be provided. For example, instead of having a high probability of missing a more important attack A_{ij}, the IDS can decide not to include the tasks for a less important attack A_{kl} so that there will be sufficient resources (queue space and service time) available for the tasks detecting A_{ij}. Decreasing $\mathcal{V}$ by $\mathcal{V}_{kl}$ rather than $\mathcal{V}_{ij}$ is a better solution because $\mathcal{V}_{kl} < \mathcal{V}_{ij}$.

Performance adaptation can be defined as the process of dynamically reconfiguring an IDS to provide the optimal value given the current run-time constraints. Note that performance adaptation cannot prevent audit data dropping or prolonged detection delay caused by stress or overload attacks, as long as the IDS has limited resources (e.g., bounded queue size) and has no control over the volume and speed of the audit data stream. The purpose of performance adaptation is essentially to manage the risk better. That is, instead of having no control over how its performance is degraded (i.e., no control over which attack will be missed) when stressed or under overload attacks, the IDS can *quickly* reconfigure to provide the best detection value under the new conditions.

Performance adaptation relies on *performance monitoring* in run-time to detect the conditions (e.g., "stress") that cause performance degradation and to measure the parameter values needed for solving the optimization problem (4.5). Chapter 6, will discuss the performance monitoring and adaptation techniques and their implementation in prototype real-time IDSs.

4.5 Practical Considerations

For rules derived from anomaly detection schemes, β_{ij} and α_{ij} can be estimated using suitable training data sets. Misuse detection rules for well defined attacks will have well understood behavior, e.g., some may even have $\beta_{ij} = \alpha_{ij} = 0$. π_i can be estimated on the basis of typical traffic statistics, and can updated periodically on basis of traffic measurements. t_{ij} can be measured by controlled experiments. The instantaneous values of m_i reflect the traffic mixture of the incoming packets. In practice, the mean value of m_i can be selected within a suitable time window. A site-specific risk analysis can provide the initial p_{ij} values, which can then be updated according to traffic and attack history. Scenario analysis (see Section 6.3) can use information on attacks detected thus far to predict the likely forthcoming attack(s) R_{ij} along with its p'_{ij}. We can use p'_{ij} as the updated (posterior) probability in place of p_{ij}. Note that as discussed in Section 4.3, the performance optimization problem allows ranges in parameters, thus the parameters need not be measured in absolute values. This relaxation should significantly simplify the measurement tasks.

Estimating the costs $\mathcal{C}_{ij}^{\beta}$ and $\mathcal{C}_{ij}^{\alpha}$ also requires site-specific risk analysis. Although it is difficult to measure exact costs, we can still learn the relative ordering of intrusions in terms of their risks (or "damage cost") [6, 17, 3]. One needs to first define a site-specific *attack taxonomy*. The damage cost of various attacks can be estimated according to factors like

- Criticality of the target: If the target is an organization's DNS server or web server, it is more critical than when the target is a normal host. [17, 13].
- Lethality of the attack: Attacks that give root access to the remote attacker is more serious than a Denial of Service attack, which in turn is more sever than reconnaissance probes.
- Other factors: Other factors that decide the damage cost is whether all the hosts or servers in an organization is immune to a particular attack or is vulnerable, depending on how recent the operating system and the patch level is.

A false alarm cost can be the penalty if an automated response is used. For example, if a normal user session is terminated, then the cost can be the same as a DoS damage

cost [13]. If an investigation is initiated, it can be the labor cost involved (wasted). Again, we can define the site-specific relative scales of false alarm costs. As discussion in a later chapter will show, since the main purpose of cost-benefit analysis is to achieve performance adaptation under resource constraint, such relative scales (not exact numbers) are sufficient for determining which intrusion detection tasks should be given higher priorities.

The next chapter describes the test bed and environment where we tested the statically configured as well as the adaptive IDS. Also the results of the experiments with statically configured IDS is shown.

Chapter 5

Experiments with statically configured IDS

This chapter explains the experiments performed to study the behavior of statically configured IDSs under a stressed or overloaded scenario. The experiments were done using Snort [24] Version 1.8.6 as well as Bro [20] Version 0.7a90. The chapter starts with a brief overview of the architecture of both Bro and Snort, and their configurations for the experiments. It also describes the test bed used and the background traffic profile. Finally, the different types of overload attacks and their traffic profile is explained.

5.1 Snort

Snort is a light weight network IDS and can be classified as a signature based IDS, although it does some minimal protocol analysis. The architecture of snort consists of three sub-systems: the packet decoder, the detection engine, and the alert subsystem. Snort depends on libpcap, a packet capture library, for sniffing the packets off the wire.

- Packet decoder: This subsystem deals with the decoding of the different headers a packet might have, starting with the data link layer header, IP header, transport layer header. Another feature of Snort is its architecture that supports plug-in modules. Various functionality like IP fragmentation reassembly, TCP stream reassembly, HTTP decoding, Telnet decoding are provided as preprocessor plug-ins.

- Detection engine: Snort has a two dimensional linked list for the detection module.

One dimension of the list matches a packet with respect to its IP addresses and ports. This set of list nodes are called Rule Tree Nodes or RTNs. The other dimension tries to match based on other attributes like TCP flags, Payload size, Payload content etc. This set of list nodes are called Optional Tree Nodes or OTNs. The various detection rules or attack signatures are parsed at the time of initialization, and the rule tree or linked list is populated. At run-time each packet after passing through the decoder module goes through the list of RTNs, and if there is a match on a particular RTN, the corresponding list of OTNs is traversed for a complete match. A complete match triggers an alert and specifies the action specified in the rule.

- Alert and Logging system: This system does the various logging and alerting functions. These are also included as plug-in modules called output plug-ins. Different output plug-ins like logging the Alerts in text format, logging into a SQL database, sending winpopup messages etc., are available.

5.1.1 Snort configuration used

The experiment setup has Snort version 1.8.6 with libpcap-0.6.2 on OpenBSD 2.9. A subset of the latest rule set of the default rule set (277 rules) coming with the distribution was used. The subset included exploit.rules, ftp.rules, telnet.rules, ddos.rules, web-iis.rules, dns.rules, web-attacks.rules, misc.rules. The rules files were latest rules as of 14th March 2002. The important preprocessors (i.e. `frag2`, `stream4`, `portscan`, `http-decode`, `unidecode`, `rpc-decode`, `telnet-decode`)were also activated.

Snort was run with the recommended options "`snort -c snort.conf -b -A fast''` to make it run as fast as possible. These options makes the logging to be in binary format, which is the most efficient.

5.2 Bro

Bro is also a real time NIDS. It is event based rather than packet based. Bro performs full packet re-assembly, connection stateful analysis, and even keystroke editing. Bro is divided into an event engine and a policy interpreter modules. Similar to Snort, Bro also depends on libpcap library for sniffing the packet off the wire. The "event engine" is responsible for processing packets and generate "events", the "policy script interpreter" is

responsible for using the site-specific intrusion detection logics, coded as "event-handlers" in the C-like Bro policy language, to analyze the events. The user can decide which all event handlers are to be loaded using a configuration file. By carefully implementing the event-handlers (i.e., determining what is an "event" and how to analyze it), and setting the packet filters to decide what kind of audit data should be captured, one can customize Bro to perform only the "important" tasks and as efficiently as possible.

5.2.1 Bro configuration used

Bro version 0.7a90 with libpcap 0.6.2 on OpenBSD 2.9 is used for the experiments. Bro was run with the policy scripts that come with the distribution, plus a few additional rules to detect the test attacks. The configuration file loaded the `http`, `udp`, `icmp`, `ftp`, `telnet` event handlers. According to the loaded event handlers specified in the configuration file, Bro sets the packet filter at the `libpcap` layer, thereby restricting the network traffic that will be processed by its event engine. The filter can also be specified in command line when starting up Bro. For example, in the experiments, "(`tcp`[13]&0x7!=0) or (port `ftp`) or (port `telnet`) or (dst port 80 or dst port 8080) or (`udp` port 53) or (`icmp`)" was used, to specify that only certain `tcp` packets (`SYN`, `FIN`, `RST` packets), or `ftp`, `telnet`, `http`, `DNS` (`udp`), or `icmp` packets are to be captured, thereby limiting the traffic by a significant amount. An adversary can still overload Bro by sending a huge amount of traffic that matches the filter. The overload situation is more severe if the packets also result in Bro events, thus overwhelming not only the packet processing and event engine level but also the policy interpreter level.

5.3 Experiment Testbed

The experiments were conducted using LARIAT [26], an IDS testbed used in the 1999 DARPA ID evaluation. LARIAT provides a configurable test environment where ID modules can be "plugged" in the testbed to capture audit data and invoke response. The LARIAT testbed setup is shown in figure 7.1. It provides many ways to configure background traffic and attack generation, and facilitates repeatable controlled experiments. LARIAT software was obtained from MIT Lincoln Lab. The testbed, with several traffic generators, can produce (simulate) intranet and (both in-bound out-bound) Internet traffic. The LARIAT testbed helps to simulate (thousands of) virtual hosts and virtual sessions,

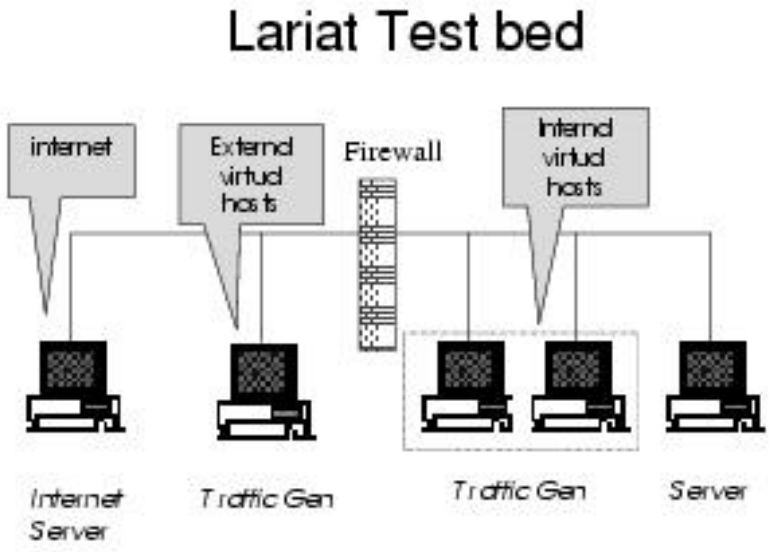

Figure 5.1: Lariat Testbed

and create high-speed and high-volume network traffic to test the algorithms and prototype IDS. The IDS was run on an Intel Pentium-3, 1 GHz processor, 512 MBRAM, a built-in Intel network interface card, and two hard disks of size 30 GB and 24 GB respectively.

5.4 Background traffic

The background traffic in the experiments was generated using LARIAT scripts based on traffic profiles. The background traffic profile is a tcp dominated one, having 97% tcp, 2% udp and 1% icmp.

Figures 5.2 shows the traffic profile in terms of connections per second. As can be observed the http traffic has the maximum connections per second. According to the bytes per second, tcp traffic occupied 99% of the data and udp,icmp sharing the 1%.

The importance is on how the IDS behaves when the traffic profile changes due to traffic fluctuations or under overload attacks. The traffic profile described serves as the background traffic and represents a tcp dominated network. The traffic profile varies widely

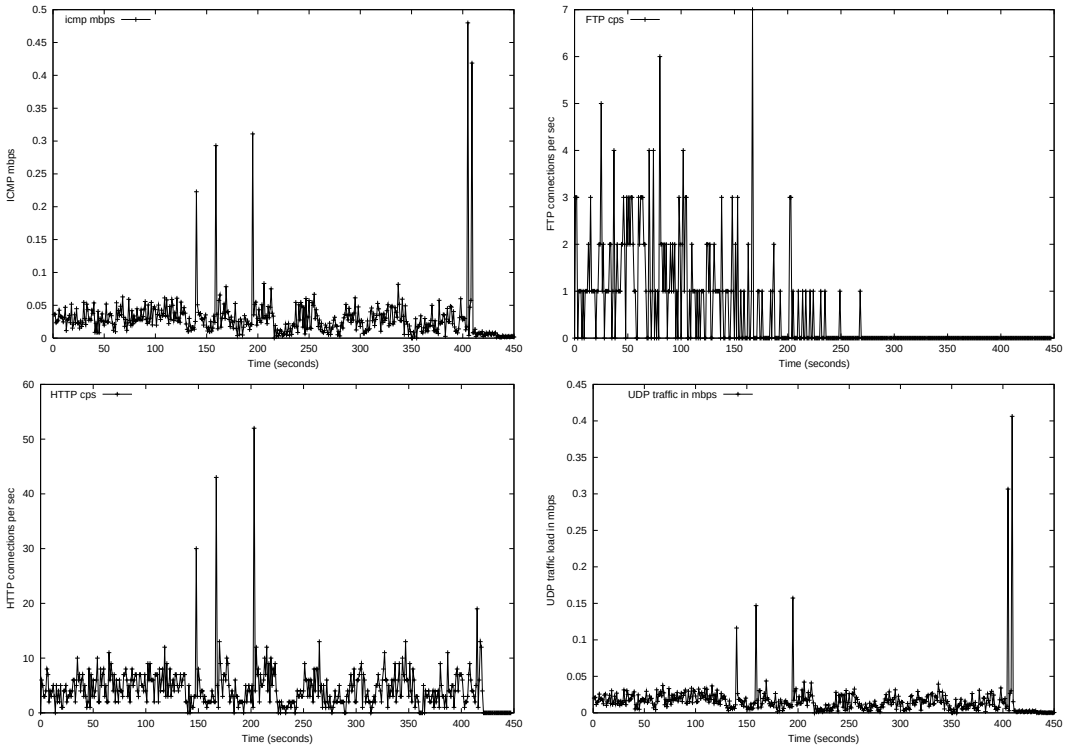

Figure 5.2: Traffic profile

over networks but usually they are tcp dominated.

5.5 Overload attacks

The overload attack experiments performed on the IDSs were done in two stages: in the first stage, the IDS is overloaded; and in the second stage a critical attack is launched. This attack which would have otherwise been detected will be missed by the IDS with a non-negligible probability. There is an inherent uncertainty from the attacker's point of view whether the attack will go unnoticed by the IDS or not. But, by making some intelligent guesses via trial-and-error, a determined attacker can still overload the IDS.

There are multiple ways to overload a resource intensive IDS, each targeting different bottlenecks in an IDS. Accordingly multiple experiments are performed showing the IDS behavior in each case.

- Traffic flooding: Figure 5.3 shows the profile of the traffic used to overload the IDS.

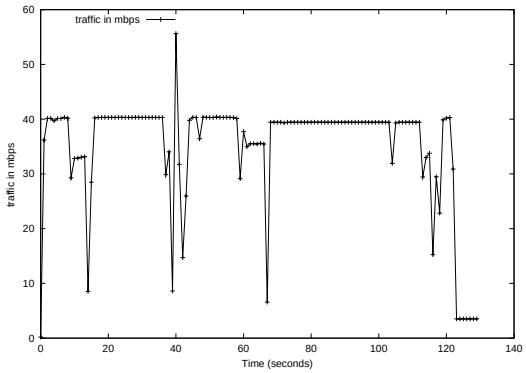

Figure 5.3: UDP flooding to overload the IDS

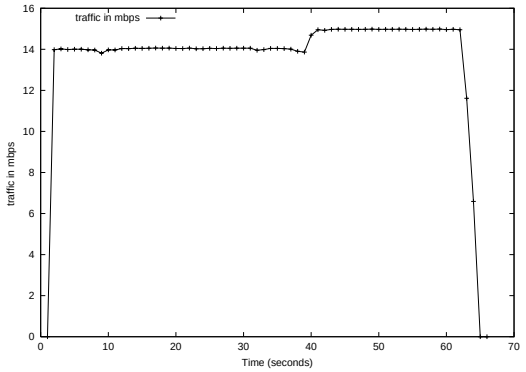

Figure 5.4: Ping flood to overload the IDS

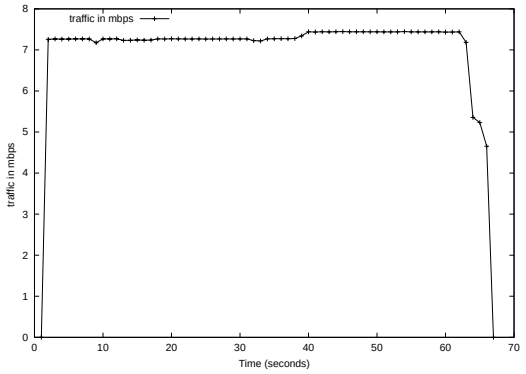

Figure 5.5: Overload attack targeting database bottleneck

Here the attacker tries to exhaust the CPU resource. This flood consisted of a (spikes) combination of `udp` packets (to `DNS` port) and `udp` packets to a user port.

- Alert flood: This method targets the alert mechanism bottleneck. The attacker has to make an intelligent guess as to whether this will cause the IDS to log an alert to the secondary storage. This method requires much less traffic load as shown by figure 5.4. The aim of the attacker is to send some nuisance attacks and increase the frequency of IDS logging.

- Database bottleneck: IDSs that maintain state information for TCP connections or for IP fragmentation reassembly can be subjected to this attack. Figure 5.5 shows the profile for this overload attack. Fragmented IP packets were sent with varying combinations of source IP address, and fragmentation id. Care was taken not to send the whole fragment set so that the IDS will keep the fragments in memory expecting to get the fragments needed to reassemble. As the number of fragments with different source IP, destination IP, fragmentation id, and protocol are sent, the insertion of another set into the database and accesses become a bottleneck.

Each of the above overload attack was followed by the attacker doing a critical attack. WEB-IIS CMD.EXE attack described below was used in our experiments.

This kind of scenario works for the attacker especially when the attack involves very few packets, for example a buffer overflow exploit or the `WEB-IIS CMD.EXE` attack[1]. On the contrary, attacks like port-scan or guess-password that involve many more packets are more likely to be detected by the IDS even when it is dropping some packets. Note that overloading need not always be intentional or malicious. An IDS monitoring a heavy network can be stressed at peak times. The attack packets were sent during the traffic surge. All the above traffic were captured using `tcpdump` and replayed using `tcpreplay`[2] version 1.1; thereby ensuring exactly the same traffic conditions for all experiments.

5.6 Experiment results

This section shows the results for each of the different scenarios described in 5.5.

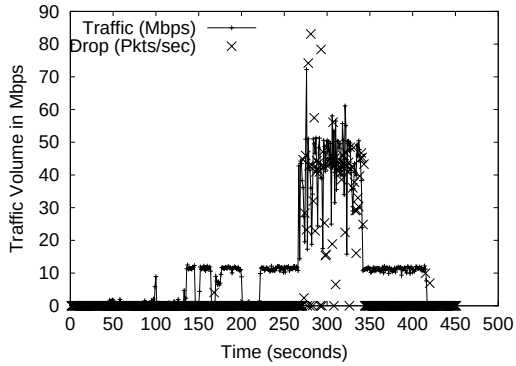

Figure 5.6: Performance of IDS under stress (attack category 1): Bro

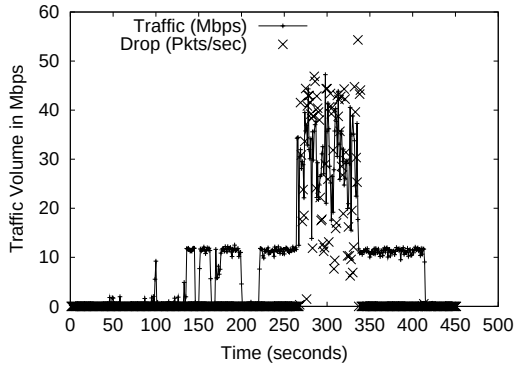

Figure 5.7: Performance of IDS under stress (attack category 1): Snort

5.6.1 Attack scenario using udp flooding

Figures 5.7 and 5.6, the results show that both Snort and Bro are stressed or overloaded when the attack method 1 is used. When the traffic volume is increased to a certain point ($>$ 40 Mbps) both IDSs are overwhelmed and drop packets. While Snort detected approximately 10% (2 out of 20 exploit packets sent) of the `WEB-IIS CMD.EXE` attacks that were launched during the flooding, Bro detected 20% (4 out of 20) of them. Both the IDSs were able to detect 100% of the exploit attempts when the traffic load was low ($<$ 10 Mbps).

[1]This is a single packet attack. The attacker sends a malicious `GET` request to a Microsoft IIS Server. The request is as follows: "`GET/scripts/..%5c%5c../winnt/system32/cmd.exe?/c+dir`". See http://www.cert.org/advisories/CA-2001-26.html for details.

[2]See http://tcpreplay.sourceforge.net/ for details.

5.6.2 Attack scenario targeting alert channel bottleneck

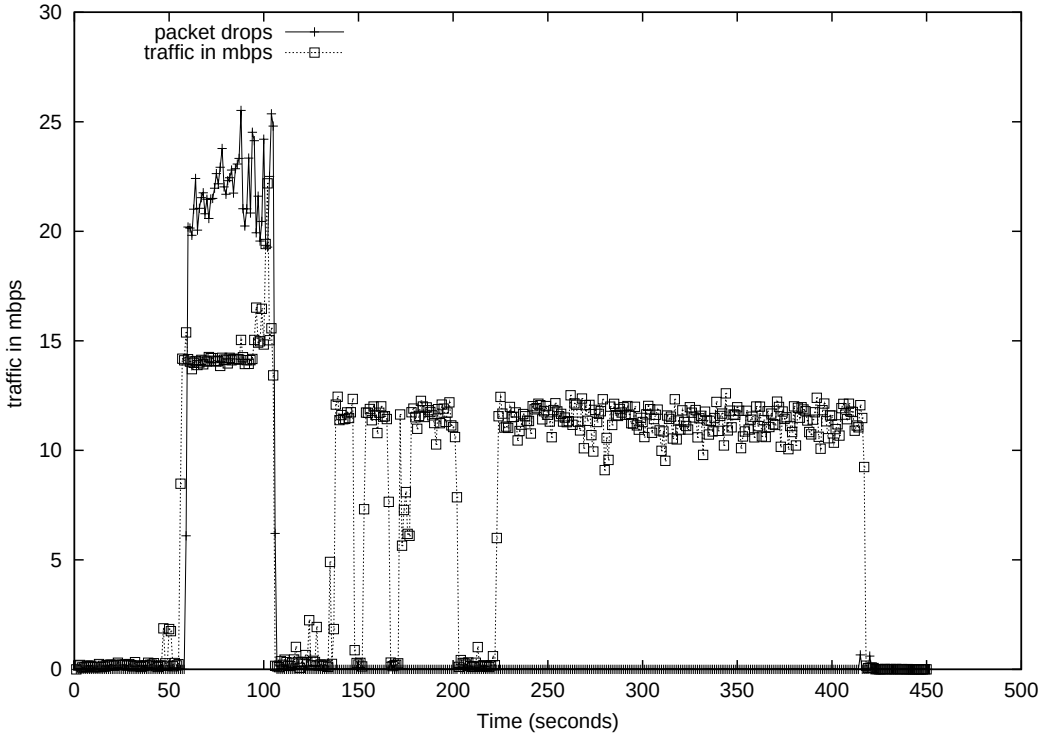

Figure 5.8: Performance of IDS under stress (attack category 2): Bro

Figures 5.8 show the behavior of statically configured Bro to the attack described in 5.5. A high frequency of "Ping flood" alerts were generated and the report was logged each time to the disk. This being a very time consuming operation overwhelmed the IDS. It can be seen from the figure that IDS dropped a lot of packets during the icmp flood, which was between t = 55 secs to t = 110 secs. Similar to the previous scenario, 20 WEB-IIS CMD.EXE attacks were attempted and in this case Bro detected none of them. This experiment and the next was done only for the Bro IDS.

5.6.3 Attack scenario targeting database channel

This attack targets the database bottleneck of the IDS. The attack profile and description is given in 5.5. The attack is done at time t = 50. The number of incomplete fragments increases with time as the attack progresses and this stresses the database ac-

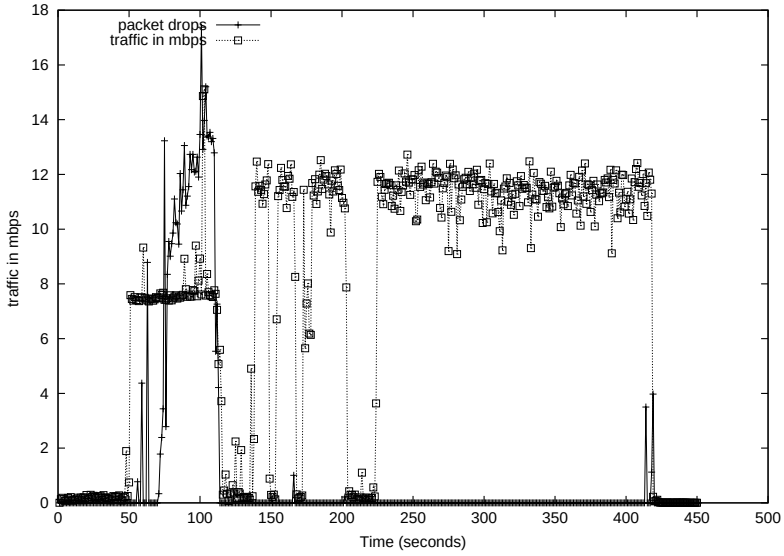

Figure 5.9: Performance of IDS under stress (attack category 3): Bro

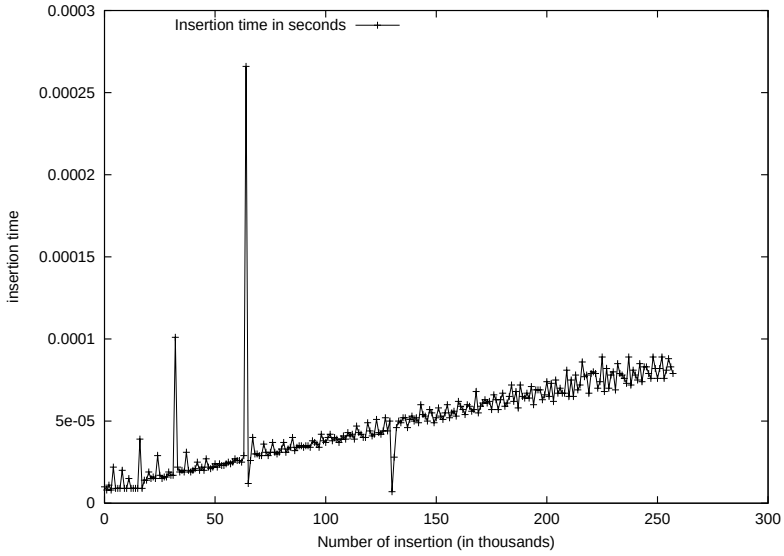

Figure 5.10: Time for insertion to database

cesses, for example, insertion. As can be seen from the figure 5.9, the drop starts slowly after the attack begins and continues to increase.

A benchmarking was done on the IDS to see how the insertion time was affected

due to this traffic. Figure 5.10 shows the details. It can be noted that as the number of insertions increases, the time for insertion increases. This overloads the IDS.

The WEB-IIS CMD.EXE attack was attempted 20 times along with this attack also. Bro was able to detect 1 out of 20 attempts. This behavior is due to the fact that the IDS takes sometime to get overloaded in this attack as compared to other scenarios.

The next chapter concentrates on the approach followed to develop performance adaptive IDS architecture.

Chapter 6

Performance Adaptation Architecture

In this chapter, we discuss how to enable real-time IDS to provide performance adaptation. We describe prototype system models.

An adaptive IDS can include multiple intrusion detection (ID) modules, performing increasingly more complex and more time-consuming analysis, and sharing the IDS workload. For example, a front-end module performs data gathering, pre-processing (e.g., packet re-assembly), and as much of the detection work as possible in real-time. A back-end module may not have stringent real-time requirement because, for example, it uses pre-processed audit data (sent from the front-end) to analyze attack trends. Its predictions on forthcoming attacks can be used to help configure the IDS.

The front-end (real-time) module needs to provide performance adaptation. Whenever it is stressed or overloaded, it computes a new optimal IDS configuration according to the new operation conditions (see Problem (4.5)). The reconfiguration deactivates some (less critical) tasks (e.g., port-scan analysis) and/or cease to capture some events. These excluded tasks can be carried out by the back-end if possible, for example, if they only require pre-processed audit data and the front-end continues to capture and process the needed audit data. We call the process of moving some analysis tasks from the front-end to the back-end *load-shedding*. It essentially allocates the limited resources (i.e., buffer space and service time) to the more critical tasks and events, thus ensuring that the front-end module can provide optimal value while satisfying the constraint $T(\mathcal{P}) \leq D_{max}$. The manager needs

to participate in monitoring the ID modules and initiating appropriate re-configuration because the ID modules can be under attacks (or even crashed) and thus may not be able to self-monitor and self-reconfigure. An "active filtering" module, such as a firewall, is desirable for first dropping the obvious offending packets, and thus cutting down the data volume to the ID modules. It can also be used as "admission control", for example, to slow down the data stream (e.g., hold and delay the packets) under some extreme situations to help the ID modules keep up with the traffic.

A popular approach to manage IDS workload is to have several front-end modules and use load-balancing to "split" the traffic [29]. Our research is complimentary to IDS load-balancing. First, performance adaptation is necessary because a front-end can still be overloaded if each traffic portion (split) is in very high volume. Second, some distributed (and network-based) attacks (e.g., port-scan) may be missed due to load-balancing because the evidence gathered at each front-end module may be below the detection threshold. We then need a correlator, which is essentially a back-end module in our architecture, to detect these attacks. Third, there are complex analysis tasks that should be performed in a back-end module rather than a front-end module because of their computational time and space requirements. These tasks include attack scenario analysis (more in Section 6.3), and alarm correlation and reduction, which are considered very important and desired IDS features [27]. In our approach, the back-end can also carry out some analysis tasks shed from the front-end.

6.1 Performance Monitoring

According to Problem (4.5), the IDS can provide the expected value $\mathcal{V}(\mathcal{P})$ only when the constraint $T(\mathcal{P}) \leq D_{max}$ is satisfied. The IDS thus needs to self-monitor the run-time conditions, and reconfigure itself to operate under the (new) constraints when necessary. As discussed in Section 4.3, D_{max} should be the mean audit event inter-arrival time.

There are two approaches in monitoring $T(\mathcal{P}) \leq D_{max}$. In *internal measuring*, since the front-end ID module knows the arrival time and detection time of each audit event, it can compute both D_{max} and $T(\mathcal{P})$ (and including m_i, t_{ij}, π_i, and p_{ij}) as a moving average. Alternatively, to avoid the overhead, the front-end can periodically check (e.g., via `libpcap`) whether it is dropping audit events, and if so, conclude that it needs to reconfigure.

In *external testing*, the manager periodically sends out a simulated attack that contains an event marked "attack-simulation". The front-end, upon "detecting" this simulated attack, is required to reply to the manager the $T(\mathcal{P})$ value along with the sequence number of the simulated attack. The manager can detect the condition where the returned $T(\mathcal{P})$ is out of bound (according to historical data), and thus concluding that the front-end is overloaded. If the manager receives no reply, it concludes that the front-end is at a "fault" state (e.g., crashed due to crash attacks [20], or an infinite-loop due to implementation errors), and can take immediate action such as activating another (replacement) front-end module.

6.2 Dynamic Reconfiguration

As described in Section 4.1, an IDS configuration is characterized by its collection of run-time analysis tasks. Although an IDS may have a very comprehensive set of tasks that it can use, its optimal configuration, that is, the solution to Problem (4.5), may include only a subset of these tasks because of run-time constraints. When performance adaptation is enabled in the IDS, this subset (the active tasks) is dynamic, that is, re-computed whenever necessary, rather than static. An implementation of the dynamic task set is to equip the ID modules with a common and complete set of analysis tasks, and have non-overlapping bit masks specifying which tasks are activated at each module.

In Section 4.5, we discussed the practical considerations in measuring the parameters needed to compute the optimal IDS configuration. We can use some heuristics to improve the parameter estimations. The back-end can perform attack scenario analysis (will be discussed in Section 6.3) and supply p'_{ij}, the probability of an attack given the traffic and attack conditions seen thus far. We can use p'_{ij} as the updated (posterior) probability of intrusion in place of p_{ij} when computing $\mathcal{V}(\mathcal{P})$ and $T(\mathcal{P})$. However, we need to avoid being fooled by an intelligent attacker who tries to divert IDS resource from his intended attacks, e.g, by first launching (nuisance) attacks that seem to lead to some other possible attacks so that they will have artificially (and falsely) much higher probabilities than his intended one(s). One solution is to *always* capture audit data and perform analysis tasks for the critical services. This is equivalent to always setting the values of these tasks the highest and their required time the smallest.

6.3 Scenario Analysis

A scenario is a sequence of related attacks that together accomplish a malicious end-goal. We can use scenario analysis to predict the likely forthcoming intrusions to make better load-shedding decisions.

We can use site-specific threat models to form a base set of known scenarios. A *scenario graph* is a directed graph where an edge from node a_i to a_j labeled with p'_{ij} and condition(s) $cond_{ij}$ specifies that after a_i occurs, a_j will occur next with probability p'_{ij} if $cond_{ij}$ is true. A path specifies an attack scenario. In run-time, each reported attack is described by a set of attributes: name (type), time stamp, target_IP, target_port, etc. The back-end "attaches" each attack to a node in the network topology graph using the target attributes, and examines whether the attack is part of an existing scenario(s) there. Based on the currently recognized (partial) scenarios, the back-end reports to the front-end and the manager the possible attacks, their probabilities, and their likely targets. The attack and probability information are used to compute $\mathcal{V}(\mathcal{P})$ and $\mathcal{T}(\mathcal{P})$ for load-shedding decisions (see Section 6.2). The target information is very useful to determine (if necessary) what portion of the audit data the front-end can stop capturing. We have a basic scenario analysis module functioning, and are actively studying how to automatically update scenario information and discover new scenarios.

The implemetation details of our approach is covered in the next chapter, covering changes and modifications done to both Bro and Snort IDSs to make them adaptive.

Chapter 7

Prototype System

We next describe the prototype systems that were derived from Bro and Snort. Our main goals here are to see how performance monitoring and dynamic configuration mechanisms can be built into an IDS, and how such adaptive IDS performs in an overload situation.

Adaptive Bro

Our adaptive IDS comprises of a front-end IDS, a back-end IDS and a manager module. We use Bro version 0.7a90 (on OpenBSD 2.9) with our modifications as the front-end IDS. The back-end module runs on a different machine and is connected to the front-end on a private network. The manager runs on a third machine and is on the same network that Bro is monitoring. Figure 7.1 shows the architecture of the adaptive IDS model. The front-end IDS monitors the network and does the real-time detection by analyzing all the traffic entering through the firewall. Front-end sends the performance information as well as alert and reports periodically to the manager. Audit records and event details are passed to the backend for extra analysis and for scenario analysis.

We modified Bro in two main areas. The first is in adding bookkeeping functions for the purpose of performance monitoring. Note that a Bro "event" is different from an "audit event" (an audit record arriving at IDS event queue) described in Section 4.1. The latter is equivalent to a packet. In Bro, events are generated from the processing of packets, and intrusion analysis (i.e., rule checking and logging) is performed on events rather than on packets. We discussed the constraint $T(\mathcal{P}) \leq D_{max}$ in Section 4.3. $T(\mathcal{P})$ is equivalent

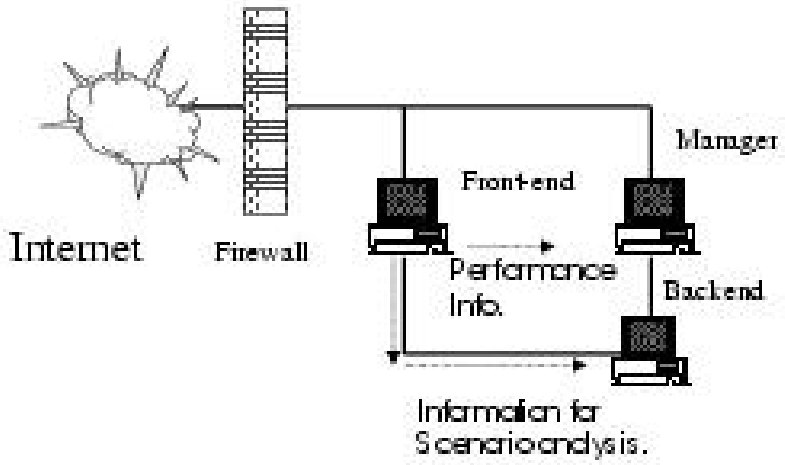

Figure 7.1: Components of Adaptive IDS architecture

to the expected packet service time in Bro, and D_{max} is equivalent to the mean inter-packet arrival time. Clearly, this constraint must still hold for Bro, otherwise there will be packet dropping (and the quality of event data will suffer) and detection performance can suffer. To accommodate the notion of event-level (versus of packet-level) analysis in Bro, we use $T'(\mathcal{P})$ to represent expected event service time, which is the interval between the arrival of the first packet of the event to the completion of analysis of the last packet of the event (also the completion of the event). We use D'_{max} to represent the mean inter-event arrival (or generation) time. For Bro events, we should use the more meaningful constraint $T'(\mathcal{P}) \leq D'_{max}$. It is easy to see that using $T'(\mathcal{P}) \leq D_{max}$ is incorrect, and $T'(\mathcal{P}) > D'_{max}$ will likely lead to $T(\mathcal{P}) > D_{max}$.

Our Adaptive Bro thus has the following measurements: number of packets received per second, number of packets dropped per second, mean inter-event arrival time, and a counter of each event. The packets received and dropped is available from the `libpcap` `pcap_stats` function and is already implemented in the Bro `HeartBeat` function. The interval between two heart beats can be configured. We used 1 second in our experiments. By recording the number of events generated within a time interval (which is 0.1 seconds), mean inter-event arrival time is computed as an average. Bro initiates reconfiguration in two cases: if it detects that there are dropped packets; or if it discovers that $T'(\mathcal{P}) > D'_{max}$.

The second area of changes to Bro is adding dynamic reconfiguration mechanisms. Recall that the process of reconfiguration is to then compute a new optimal solution to Problem (4.5) according to the new run-time constraints, and then deactivate some analysis tasks and/or cease to capture certain audit data types according to the newly computed configuration. The reconfiguration is implemented as a greedy algorithm. After ordering the tasks in decreasing order of value/cost, the algorithm does the selection in a "greedy" way. Unlike the classical knapsack approach, all the combinations are not tried, which makes it computationally less expensive but may not come up with the most optimal solution. The parameters associated with the event-level analysis tasks are initially measured using benchmark experiments and stored in a system configuration file. For example, the service time for a specific event is the average time taken by Bro to process packets, generate the event, and analyze the event (e.g., match it against rules). The parameters are then loaded in an array in Bro start-time, and can be dynamically updated. For example, π_i and m_i are measured as moving averages in run-time, and p'_{ij} from the scenario analysis function in the back-end can replace p_{ij}. The computed configuration is represented as an array of flags (Bro script variables). These flags are checked before the event analysis tasks (handlers) are invoked. If all event analysis tasks for an audit type are disabled, then `libpcap` filter is also reset to cease capturing such data. Since compiling and loading a new filter at the `libpcap` layer incurs significant delay, we modified Bro to keep a set of pre-compiled filters and load them when necessary. This reduces the time taken for reconfiguration. Changing the filter applied at the libpcap layer to be a more specific one controls the amount of traffic being processed ie. controlling the m_i. This is the most effective way since the control is at the lower most layer of the IDS system. Also, when changing packet filters, the `pcap_setfilter` function invokes the `ioctl` kernel function. It turns out that `ioctl`, while changing the filters, clears out the packets that have not been passed to the upper layer. We took out the code that clears the buffer to avoid losing those packets that might match the new filter. Finally, Bro has an option to store (remember) the "default" configuration, the start-up configuration which is considered as the 'optimal" or desirable one under normal situations, so that if it was reconfigured and has been stable (no need to reconfigure again) for several heart beats (in our experiments, we used 10), it can switch back to run the default configuration.

We briefly describe other modules in our system. The main functions of the manager are to collect statistics and intrusion reports from the Bro and the back-end, and

create detailed logs and alerts. It sends a test periodically to Bro to measure delay. Bro also sends the performance measurements (e.g., the numbers of packet received and drop) every heart beat. If the manager does not receive Bro performance measurements and or a reply from its test for a time threshold, it raises an alarm (to security staff) that Bro has probably crashed. The policies on the firewall can be dynamically configured by the front-end. For example, it can send RST segments to either end of a misbehaving connection, or block an ip-address and/or port. It can also delay packets when instructed. The main functions of the back-end module include: sharing analysis load, for example, probe (scan) detection shed from the front-end, and performing attack scenario analysis. The scenario analysis module is in its primitive stage of development and is not a contribution of this thesis.

Adaptive Snort

We also implemented an adaptive IDS using Snort version 1.8.6 with the latest rule set, and with `libpcap` version 0.6.2 and OpenBSD version 2.9. Unlike Bro, Snort applies intrusion detection rules on packet data directly rather than on "events" extracted from packet data. Snort supports "plug-ins", which can be pre-processors (e.g., fragmentation reassembly) or detection rules. Snort is thus more loosely coupled and easier to customize. We wanted to study how different IDS architectures influence the implementation of performance adaptation mechanisms.

In Snort, packets go through first the pre-processors then the rule trees. A RuleTreeNode determines whether a packet is a "match" and hence needs to be examined by its OptTreeNodes. In Bro, we can measure service time at the Bro event level and use event service time to include preprocessing and event analysis time because each packet contributes to a Bro event. For Snort, we need to measure the service time at the packet level. There are two cases. First, for packets that match an OptTreeNode (i.e., they match or "belong to" a particular Snort rule), the service time is preprocessing time plus rule checking time (which is the time spent traversing the rule tree up to and including the OptTreeNode). In this case, we call the service time T_R and keep a measurement for each OptTreeNode (i.e., each Snort rule). Second, for packets that do not match any of the RuleTreeNodes (i.e., they do not belong to any Snort rule), the service time is the preprocessing time plus the time traversing the RuleTreeNodes. In this case, we call the service time T_P and keep a measurement for packets of each protocol: `http`, `telnet`, `ftp`, `ssh`, `finger`, `other-tcp`,

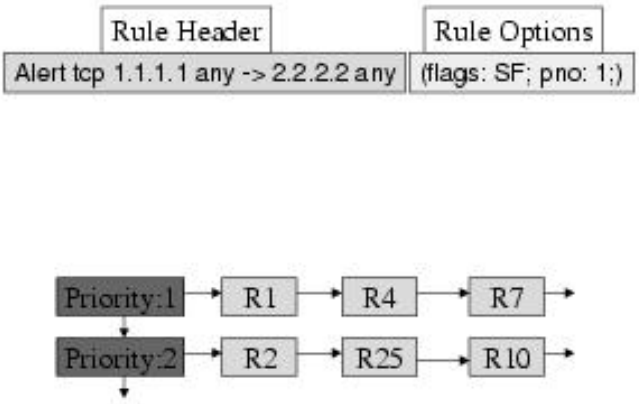

Figure 7.2: Rule structure and Priority list data structure

`icmp`, and `udp`. We need to include T_R and T_P measurements when computing an optimal Snort configuration. Since preprocessing is the main factor in T_P, we need to consider the "value" of preprocessing in addition to the values of the rules. We assign the highest value to preprocessing because it is always needed. If T_P is too high (e.g., when Snort is overloaded by packets that do not necessarily match rules), the Knapsack algorithm can output a configuration that does not include preprocessing. Such a configuration is not acceptable. In such a case, the following iterative process is used: use Knapsack algorithm to first determine what packet filters should be used (what protocols are allowed) in order to keep T_P low (e.g., half of the value in the previous iteration), using priorities among the protocols; then use Knapsack to compute a Snort configuration, considering both T_P and T_R; if a configuration including preprocessing is output, then terminate, otherwise, continue to iterate.

Figure 7.2 shows the structure of the snort rule. We added an additional option "pno" which represents the priority of the rule or importance of the corresponding attack. In order to efficiently enable and disable Snort rules without having to traverse the entire rule tree data structure, we implemented a direct access mechanism. It uses a two-dimensional linked list as shown in figure 7.2. The head nodes in one dimension are the priorities of the rules (i.e., rank orders in terms of their values), and the other dimension comprises of a list

of pointers to all the rules having the same priority. This data structure is populated when parsing the rules at Snort start-up time.

In the next chapter we discuss the results of the experiments that were done on the adaptive IDSs, both Bro and Snort.

Chapter 8

Experiments on the Prototype Systems

This chapter gives the details of the set of experiments performed on our prototype adaptive IDSs. These experiments are similar to those described in Chapter 5.

8.1 IDS configurations and parameters

The initial configurations for both the IDSs were exactly the same ones used for the statically configured Bro and Snort.

8.1.1 Initial configuration for Adaptive Bro

For Adaptive Bro, the initial configuration is to detect all of its "known" attacks, which include more than 100 detection rules on root access (e.g., `imapd` buffer-overflow), user access (e.g., `PHF`), DoS (e.g., `smurf`, `syn-flood`), and probes (e.g., `portsweep`). The initial `libpcap` filters were set to be "(`tcp`[13]&0x7!=0) or (port `ftp`) or (port `telnet`) or (dst port 80 or dst port 8080) or (port `imap`) or (`udp` port 53) or (`icmp`)" as before.

8.1.2 Initial configuration for Adaptive Snort

For Adaptive Snort, the initial configuration consisted of a subset of the default rule set (277 rules) coming with the distribution. The subset included exploit.rules, ftp.rules, telnet.rules, ddos.rules, web-iis.rules, dns.rules, web-attacks.rules, misc.rules. The

rules files were latest rules as of 14th March 2002. Different priorities were given to each rule using the "pno" option described in figure 7.2. For example, exploit rules were given highest priority, followed by ftp rules and web-iis rules, followed by telnet rules. The alloting of the priority should be according to site level policies. The important preprocessors (i.e. `frag2`, `stream4`, `portscan`, `http-decode`, `unidecode`, `rpc-decode`, `telnet-decode`) were also activated.

Regarding the parameter measurements, we assigned damage costs ($\mathcal{C}_{ij}^{\beta}$) of intrusions in relative scales: 100 for root access, 50 for user access, 30 for DoS, and 2 for probing, according to analysis in [17, 13]. Since we use automatic intrusion responses (using the firewall), we assign all false alarm costs ($\mathcal{C}_{ij}^{\alpha}$) the same as the DoS damage cost. Since we do not have statistics on attack distribution yet, we assign the prior probabilities (p_{ij}) of all intrusions to be the same (effectively, 1). As mentioned above, π_i and m_i are measured in run-time.

The background traffic was also the same as used for the experiments and was replayed using `tcpreplay`.

8.2 Results of experiments with prototype IDSs

8.2.1 Attack scenario using udp flooding

The profile of the attack is shown in 5.3, the background traffic being the same.

Figure 8.1 shows the behavior of the Adaptive Bro when the overload attack using udp flood was used as in figure 5.3; the background traffic being the same. We describe some details as follows. Initially when the traffic is low, the inter-event arrival time is high and the systems can perform all the analysis tasks. When the traffic rises high, the inter-event arrival time drops low and Bro discovers that $T'(\mathcal{P}) > D'_{max}$. It then invokes Knapsack to compute a new optimal configuration for the current conditions. Also it can be noted that there were initial packet drops due to the heavy load due to the udp flood, but the quick reconfiguration avoided further packet drops. This happens at time $t = 270$. This is different from the Original Bro, as shown in Figures 5.6 and 5.7, where the situation of packet drops continues. Since the flood was caused by `udp` packets, the m_i value of `udp` increased a lot and so did the "weights" (time requirements) of their analysis tasks (see Problem (4.5)). The reconfiguration ended up dropping all tasks for `udp` and hence the

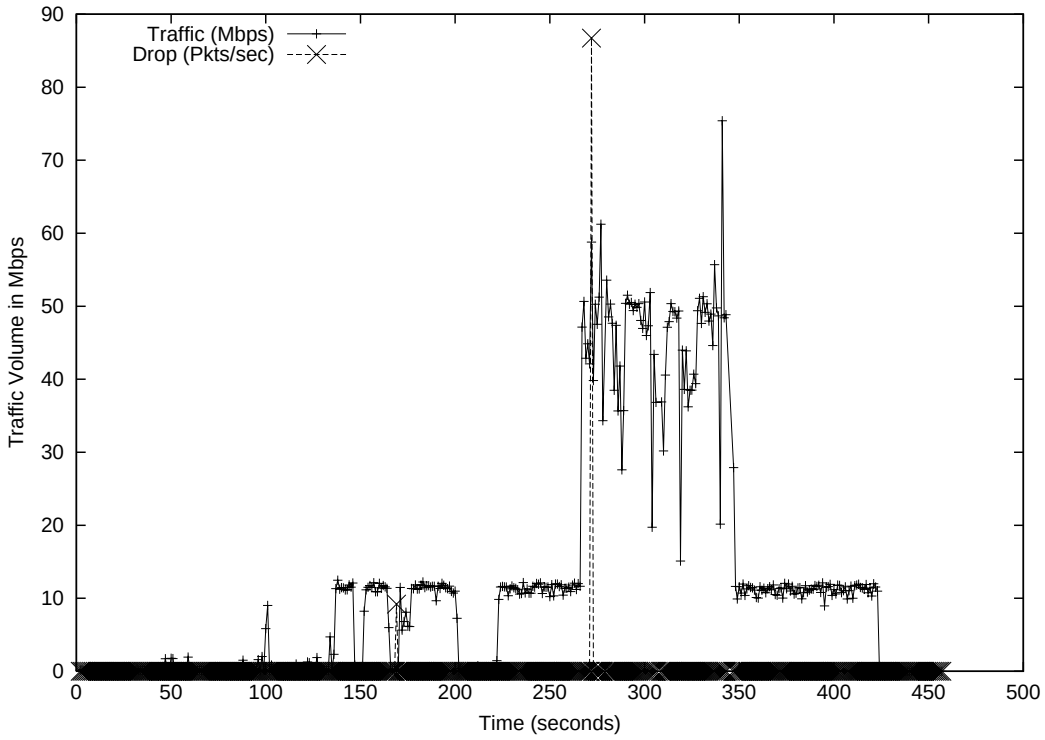

Figure 8.1: Behavior of adaptive Bro for overload attack category 1

filter for picking up `udp` traffic. Due to the reconfiguration the Adaptive Bro detected 95% of the WEB-IIS CMD.EXE attacks that were done during the flood where as the statically configured Bro was able to detect only 20% of them.

The overload attack using ping-flood is started at t = 57 seconds. WEB-IIS CMD.EXE attack being an attack that gains "root" privileges, had been given a relative damage cost 100. Therefore, by changing the configuration the adaptive IDS was able to keep its false negative rate from going low, and thereby attaining a better IDS value.

It should be noted that by dropping all the udp related rules and changing the filter, it will not be able to detect the attacks in that category, if they occur. So the IDS value is lesser than the original value it had, but the aim is to get the next optimal value. The dns rules (udp) had relatively much less damage cost compared to exploit rules like WEB-IIS CMD.EXE, using which an attacker can potentially gain "root" access on the target machine. Therefore, an optimum value for the IDS is obtained by skipping relatively lower priority detection rules and ensuring the detection of attacks which have relatively

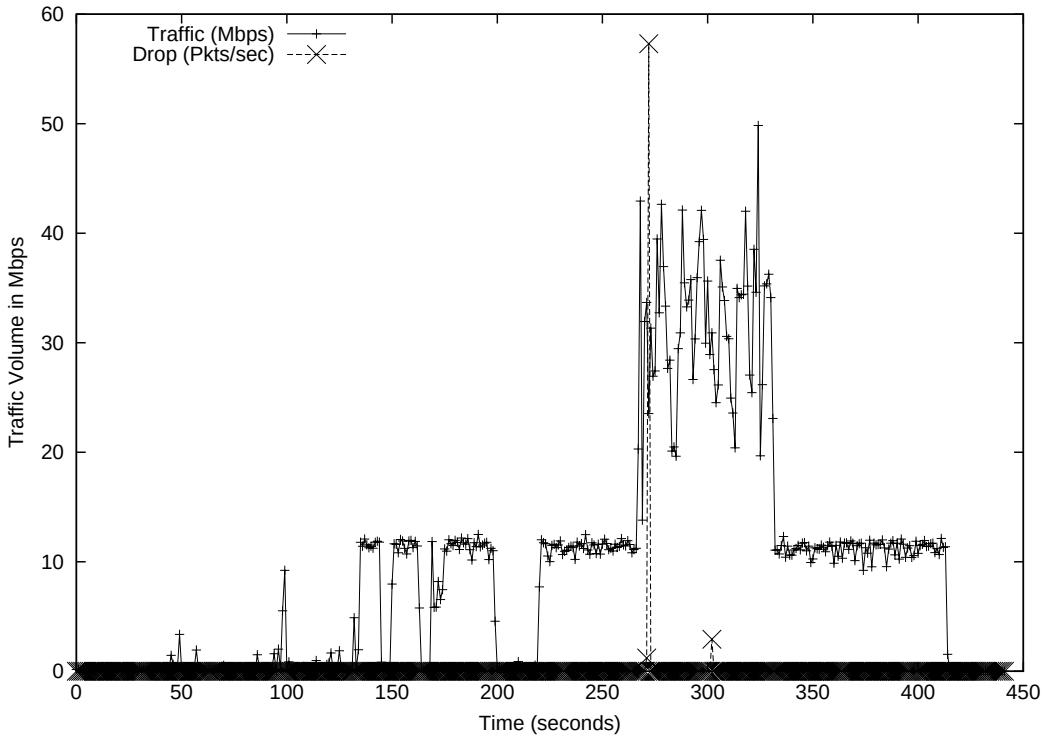

Figure 8.2: Behavior of adaptive Snort for overload attack category 1

higher priority.

Figure 8.4 shows the reduction in value contributed by the critical rule (WEB-IIS) for Original Bro. This is because of the decrease in detection rate (or increase in false negative rate) as shown in figure 8.3 . Figure 8.5 shows the reduction in the total value of the IDS. It can be noted that Adaptive Bro gives more value by giving priority to the higher priority rules and controlling the false negative rates of critical attacks. The contribution to the total value is more from http, ftp rules than from udp rules. By shedding all the udp rules and filter, value of the IDS has changed only from 56.5 to 56 as shown in figure 8.5. This is because UDP rules are given lesser relative damage costs and the π_i is also low.

The false negative rate for all rules are assumed to be affected by the same amount and that the false positive for all rules is unaffected.

Similarly, for Adaptive Snort, when the system is overloaded, it performs a quick reconfiguration (disabling `udp` rules and packet filter) to avoid further packet drops. Adaptive Snort detected the WEB-IIS CMD.EXE exploits 100% of the time (20 out of 20).

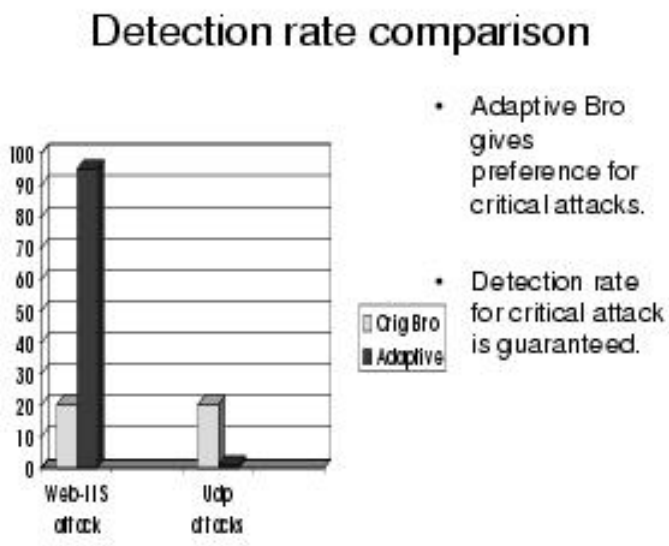

Figure 8.3: Detection rate during attack scenario 1

8.2.2 Attack scenario targeting alert channel

The profile of the attack is shown in figure 5.4 and the background traffic is the same. Figure 8.6 shows the behavior of adaptive to the overload scenario in which the attacker loads the alert channel bottleneck. There was zero drops; therefore drops curve is not included. The figure shows the variation of average inter-event-interval when the traffic fluctuates. Especially figure 8.7 shows the details from time t = 48 secs to t = 67 secs. At time t = 59, the average inter-event-interval, D_{max} became very low and the equation 4.5 was not satisfied and the knapsack algorithm was invoked. A part of the icmp rules including ping-flood, smurf, ping-scan was removed by the knapsack.

All the 20 WEB-IIS CMD.EXE attacks were detected by adaptive Bro during this experiment; thereby having a detection rate of 100%. Statically configured Bro had zero detection rate. Again we can note the selection of optimum value for the IDS by skipping relatively lower valued detection rules for ensuring the detection of attacks which have relatively higher damage costs. Ping flood and smurf comes under the DoS category and ping scan comes under reconnaissance attack category, both relatively lower than exploit attacks trying to attain “root” privilege on the target machine.

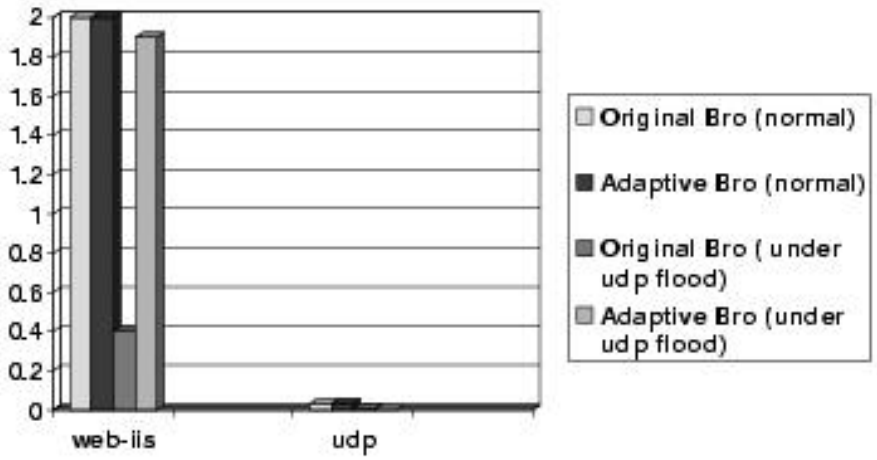

Figure 8.4: Change in value contributed by rules when IDS is overloaded

The change in detection rates for Adaptive Bro as compared to Original Bro can be seen in figure 8.8. As described in the previous experiment, the reduction in detection rate for the critical attack reduces the value of IDS for the non adaptive Bro. Since ping flood and smurf does not contribute significant amount to the value of the IDS, Adaptive Bro does not suffer much change from the original value.

8.2.3 Attack scenario targeting database bottleneck

The profile of the attack is shown in figure 5.5 and the background traffic is the same. This attack tries to stress the IDS using the database bottleneck. The traffic profile in figure 5.5 shows lot of fragmented IP packets are sent with varying combinations of source IP address and fragmentation id, so that each time the IDS makes an "insertion" to database. Figure 8.9 shows how adaptive Bro behaves in this scenario. As can be seen from figure 8.10, the insertion time increases and causes some stress and adaptive Bro drops some packets initially. The continuous increase in the number of fragmentation sets in the database triggers the reconfiguration mechanism. It changes configuration by reducing the IP fragmentation reassembly timer thereby controlling the non-stop increase of the

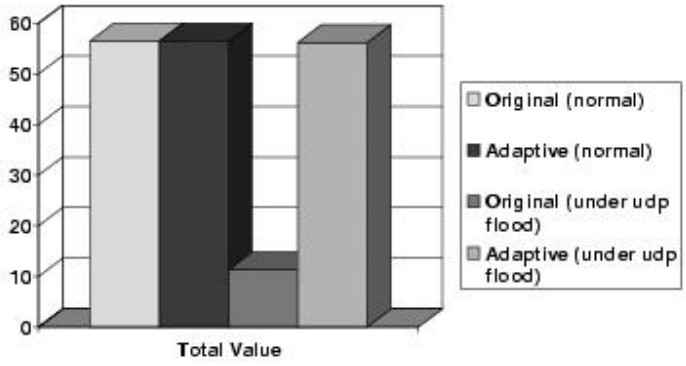

Figure 8.5: Change in Total value of the IDS (Bro)

insertion time. The timer was decreased by 50% (120 seconds to 60 seconds). This caused the insertion time to come down to normal value after sometime. The fragmentation timer is defined in the Bro configuration file and is implemented as a bro policy script variable. This value is changed by the dynamic reconfiguration module. After a specified interval (for example, 60 seconds), the value is changed back to original value.

A total of 19 out of 20 WEB-IIS CMD.EXE attacks were detected by adaptive Bro during this experiment; thereby having a detection rate of 95%. Statically configured Bro had only 5% detection rate.

Adaptive bro did not change any filter or shed any rules but still there is a reduction in the value of the IDS due to the change in the timer value. Each rule depends on fragmentation reassembly and therefore contributes some cost to it.

By reducing the timer to 50%, the fragments in memory which are not reassembled, will be timed out sooner by a factor of 2. The insertion time correspondingly decreases. Knapsack algorithm reduces the fragmentation timer till the equation 4.5 is satisfied. The value by which the timer value is decreased is configurable. For this experiment the value of 50 was used.

8.3 Overhead of performance monitoring and reconfiguration

An important consideration when employing performance monitoring and dynamic reconfiguration mechanisms is their overheads. We used micro-benchmark experiments to compute these overheads. As for Adaptive Bro, at each "heart beat" (1 second in our experiment), on average 0.0002 seconds are spent on computing the numbers of packets received and drops, the mean inter-event arrival time, and the mean event service time, the overhead is thus .02%. We believe the overhead is acceptable. Changing filters (using pre-compiled filters) takes only an average of 0.00005 seconds each time, which is very fast. Running the Knapsack algorithm takes an average of 0.0002 seconds each time. As for Adaptive Snort, the "heart beat" function (which is a timer function invoked every 1 second), takes 0.00005 seconds. The corresponding overhead is thus 0.005%. Running the Knapsack algorithm takes approximately 0.0002 seconds. The filter change (using pre-compiled filters) takes an average of 0.00005 seconds. We believe the overheads are acceptable in both systems.

8.4 Important points while designing an Adaptive IDS

This section lists out some difficulties we faced while trying to implement adaptive behavior on Bro and Snort, and try to list the features to consider if an IDS is developed with adaptive behavior in mind.

The design goals of both Bro [19] and Snort [25] does not include this adaptive behavior. Therefore, implementing performance adaptation in them was not staightforward. Some of the difficulties we faced as well as some wish-list for a future adaptive IDS are listed as follows:

- Analysis tasks was ingrained into the IDS functionality; especially in the case of Bro. Different tasks, for example, TCP stream reassembly were not physically separate in the code. So to shed the analysis tasks was difficult. The plug-in model in Snort architecture was really good in this perspective. It is advised to have a modularized architecture, especially with the costly and important analysis tasks. So that they can be plugged-in only when needed.
- There was no way to check what percentage of the packet capture buffer was currently full. We are modifying libpcap to have such statistics for the user, so that it can be

used to decide when to change the configuration.

- Libpcap packet capture library is the standard mechanism all the current IDSs use to "sniff" the packets off the wire. This mechanism becomes a bottleneck with high speed data acquisition. The fact that libpcap is written with portability in mind compromises on the efficiency.
- It would be good to have some priorities associated with the analysis tasks as well as the rules. The configuration of the IDS should be able to specify different priorities to tasks and rules at run time, thereby controlling which tasks or rules are given more importance.

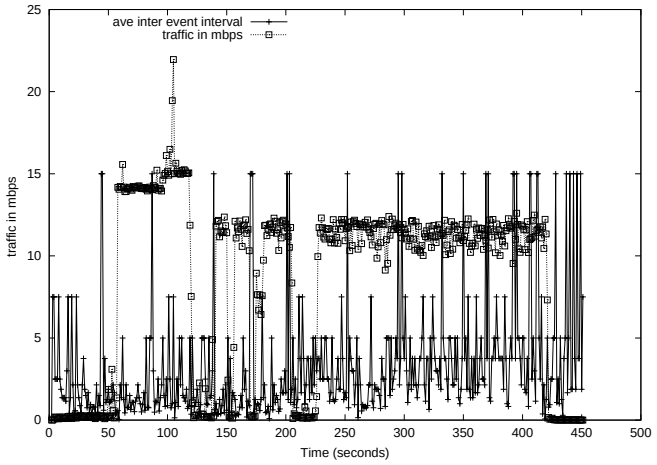

Figure 8.6: Behavior of adaptive Bro for overload attack category 2

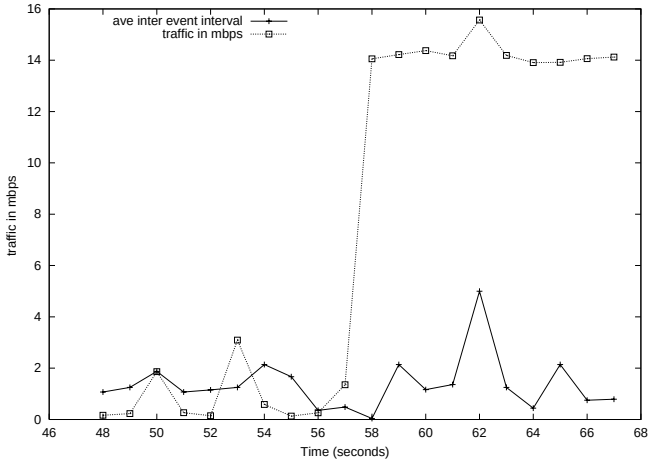

Figure 8.7: Enlarged view of time t = 48 to t = 67 seconds

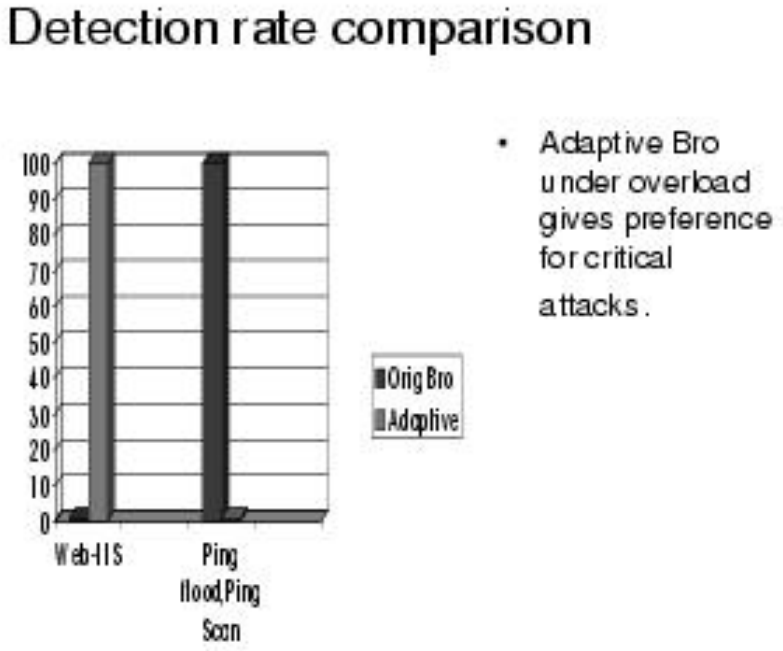

Figure 8.8: Detection rate during attack scenario 2

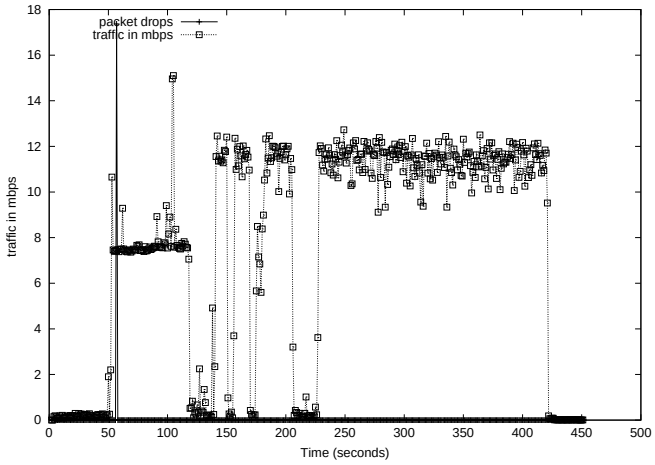

Figure 8.9: Behavior of adaptive Bro for overload attack category 3

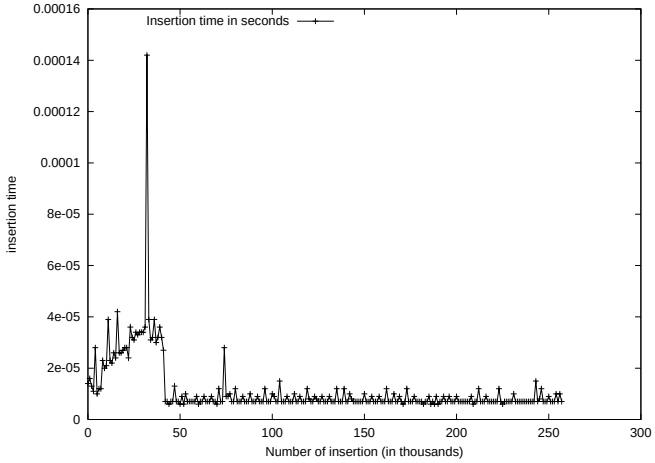

Figure 8.10: Insertion time for adaptive Bro

Chapter 9

Conclusion

Providing performance guarantee (assurance) should be the key requirement of IDSs, and security products in general. Critical attacks should be given more priority when the IDS cannot keep up with the traffic. In this thesis, we discussed an analysis of IDS performance metrics and constraints, and argued that an IDS should provide the best value under operational constraints. This is essentially an optimization problem. We showed that a statically configured IDS can be overloaded by adversaries to a point that it will miss the intended attacks with high probability. We argued that an IDS should at least achieve a weaker goal: providing performance adaptation, i.e., providing the best possible performance for the given operation environment. We discussed that in order to enable performance adaptation in real-time IDS, performance monitoring and reconfiguration mechanisms are needed. We described prototype adaptive IDSs based on Bro and Snort. Different overloading scenarios, each focussing on different performance bottlenecks of a real-time IDS were used. Results showed that IDS performing adaptation at run-time, gives optimal configuration and better performance. Packet drops are controlled thereby controlling the false negatives of important rules and getting optimal IDS value. Results from experiments thus far validate our motivation and approach. Finally, we discussed the difficulties faced and listed some of the main features a performance adaptive IDS should have.

As for future work, we will be conducting more extensive and realistic experiments. We will refine the performance monitoring and adaptation mechanisms, focusing on lowering the overheads and making them not only customizable but also dynamically configurable.

Although we were able to add performance adaptation mechanisms to Bro and

Snort, it was not without conceptual and architectural difficulties. We thus plan to follow our formal framework to design and implement an adaptive real-time IDS with built-in performance monitoring and dynamic optimization capabilities.

Bibliography

[1] Edward Amoroso. *Intrusion Detection*. Intrusion.net, 1999.

[2] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3), 2000.

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

[4] J. S. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An architecture for intrusion detection using autonomous agents. Technical report, COAST Laboratory, Department of Computer Science, Purdue University, West Lafayette, IN, 1998.

[5] J.B.D. Cabrera, W. Lee, R. K. Prasanth, L. Lewis, and R. K. Mehra. Optimization and control problems in real time intrusion detection. submitted for publication, March 2002.

[6] D. Denning. *Information Warfare and Security*. Addison Wesley, 1999.

[7] J.E. Gaffney and J. W. Ulvila. Evaluation of intrusion detectors: A decision theory approach. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[8] R. Gopalakrishna and E. H. Spafford. A framework for distributed intrusion detection using interest driven cooperating agents. In *The 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, October 2001.

[9] The NSS Group. Intrusion detection systems group test(edition 3), June 2002.

[10] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[11] L. Kleinrock. *Queuing Systems, Vol. 1: Theory*. John Wiley & Sons, Inc., 1975.

[12] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, May 2002.

[13] W. Lee, W. Fan, M. Miller, S. J. Stolfo, and E. Zadok. Toward cost-sensitive modeling for intrusion detection and response. *Journal of Computer Security*, 2001. to appear.

[14] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons Ltd., 1990.

[15] S. McCanne, C. Leres, and V. Jacobson. `libpcap`. available via anonymous ftp to ftp.ee.lbl.gov, 1994.

[16] Kevin Wiley Mike Hall. Capacity verification for high speed network intrusion detection systems, 2002.

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

[18] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity*. Prentice-Hall, Inc., 1982.

[19] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[20] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), December 1999.

[21] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore MD, October 1997.

[22] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc., January 1998. http://www.aciri.org/vern/Ptacek-Newsham-Evasion-98.ps.

[23] N. Puketza, K. Zhang, M. Chung, B. Mukherjee, and R. Olsson. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering*, 22(10), October 1996.

[24] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA Conference*, November 1999. Snort is available at http://www.snort.org.

[25] Martin Roesch. Snort - lightweight intrusion detection for networks, 1998.

[26] L. M. Rossey, R. K. Cunningham, D. J. Fried, J. C. Rabek, R. P. Lippmann, and J. W. Haines. LARIAT: Lincoln adaptable real-time information assurance testbed. In *The 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, October 2001.

[27] G. Shipley and P. Mueller. Dragon claws its way to the top. In *Network Computing*. TechWeb, August 2001.

[28] SunSoft. *SunSHIELD Basic Security Module Guide*. SunSoft, Mountain View, CA, 1995.

[29] Top Layer Networks and Internet Security Systems. Gigabit Ethernet intrusion detection solutions: Internet security systems RealSecure network sensors and top layer networks AS3502 gigabit AppSwitch performance test results and configuration notes. White Paper, July 2000.

[30] G. Vigna, R. A. Kemmerer, and P. Blix. Designing a web of highly-configurable intrusion detection sensors. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, October 2001.