Performance Adaptation in Real-Time Intrusion Detection Systems

Wenke Lee[1], João B. D. Cabrera[2], Ashley Thomas[3], Niranjan Balwalli[1], Sunmeet Saluja[1], and Yi Zhang[1]

[1] College of Computing, Georgia Institute of Technology,
801 Atlantic Drive, Atlanta, GA 30332, USA
{wenke, niranjan, sunny, yizhang}@cc.gatech.edu
http://www.cc.gatech.edu/fac/Wenke.Lee

[2] Scientific Systems Company Inc.,
500 West Cummings Park, Suite 3000, Woburn, MA 01801, USA
cabrera@ssci.com

[3] Department of Electrical and Computer Engineering,
North Carolina State University,
Raleigh, NC 27695, USA
athomas@unity.ncsu.edu

Abstract. 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 paper, 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. We show that IDS performance trade-offs can be studied as classical optimization problems. We describe an IDS architecture with multiple dynamically configured front-end and back-end detection modules and a monitor. The IDS run-time performance is measured periodically, and detection strategies and workload are configured among the detection modules according to resource constraints and cost-benefit analysis. The back-end 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.
Keywords: real-time intrusion detection, performance metrics, performance adaptation, optimization.

1 Introduction

Intrusion detection is a critical component of the defense-in-depth network security mechanisms. An intrusion detection system (IDS) monitors operating system or network activities by capturing and analyzing audit data (e.g., `BSM` [33] or `libpcap` [19] stream) to determine whether there is an attack occurring. Most systems perform *misuse detection* by pattern matching known attack behavior or effects, and some systems also employ *anomaly detection* techniques, which flag unacceptably large deviation from normal profiles as (probably) the result of an attack. A real-time IDS differs from an off-line IDS in that it tries to detect and respond to an attack in real-time (i.e., when it is unfolding).

An IDS is a "mission-critical" system, one that needs to be effective and available. More specifically, its *performance objectives* include: good detection coverage, economy in resource usage, and resilience to stress [26]. 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 [25, 23]. 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. 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.

It is well known that most current IDSs, employing only misuse detection techniques, cannot detect new attacks. Worse yet, even for known attacks, the detection performance cannot be guaranteed when the IDSs are under stress (e.g., due to high traffic volume) or are targeted by evasion, overload, and crash attacks [25, 23, 30]. Researchers are developing attack resistance techniques. Many evasion attempts can be foiled if an IDS uses stateful analysis and employs a network traffic *normalizer* [12]. Some IDSs are carefully designed to be very "light weight" or are specially configured with high-end hardware (e.g., RealSecure with AppSwitch [34]) to cope with high-speed and high-volume traffic. However, as our analysis and experiments in this paper show, as long as an IDS is *statically* configured in run-time, an intelligent adversary can overload the IDS to a point that it will miss the intended attack with high probability.

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 [10]. 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 forced to miss some intrusions (that can otherwise be detected using its "signature base"), for example, 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 argument sake all other factors, for example., attack probability, detection probability, are equal), then missing a port-scan is better than missing a buffer-overflow. In this research, we extend work on IDS cost-analysis [10, 15] to provide a framework for considering the trade-offs of IDS performance objectives, develop techniques for run-time performance measurement and monitoring, and for dynamic adaptation and reconfiguration of IDS policies and mechanisms. We currently focus our work on misuse detection systems.

The rest of the paper is organized as follows. We first analyze IDS performance issues, objectives, and discuss the need for performance adaptation. We then discuss how to enable performance adaptation in real-time IDS. We describe prototype real-time adaptive IDSs, and present experiments and results. We then compare our research with related work, and conclude the paper with a summary and a discussion on future work.

2 IDS Performance Analysis

In this section, we analyze the trade-offs in IDS performance objectives from an optimization and control perspective, and discuss the danger of static configuration and hence the need for performance adaptation. We found it necessary to introduce a more abstract formalism in order to study the problems in a general fashion. Later in the paper, we will explain how these principles can be applied in practice.

2.1 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 `telnet`, `http`, `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 `telnet` connection (and its packets) can be part of a "port-scan", and there can be a "guess-password" or "buffer-overflow" attack in it. 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,
  $p_{ij} = \mathbb{P}\mathrm{rob}(E_i \leftarrow A_{ij})$. 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, $p_{i0} = \mathbb{P}\mathrm{rob}(E_i \leftarrow A_{i0})$.
- False Alarm Cost: The cost associated with a response triggered by a false alarm that attack A_{ij} is present, denoted as $\mathcal{C}_{ij}^{\alpha}$.
- Damage Cost: The cost associated with attack A_{ij} being missed by the IDS, denoted as $\mathcal{C}_{ij}^{\beta}$.

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. We denote the tasks as R_{ij}, where $j = 1, 2, \cdots, K_i$. We say that $R_{ij} \stackrel{r}{\leftarrow} A_{ij}$ when a Detection Rule R_{ij} reports the presence of attack A_{ij} in audit event E_i. We say that $R_{ij} \stackrel{r}{\leftarrow} A_{i0}$ when R_{ij} reports that event E_i is normal. The detection rules are characterized by the following quantities:

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

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.

2.2 Performance Metrics

Expected Value The purpose of a real-time IDS is to detect intrusions and prevent damages. Instead of using mere statistical accuracy, we should evaluate an IDS according to its value (or cost-benefit). 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:

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

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

Response Time Figure 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. 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 2. That is, each event goes through a sequence of analysis tasks. The process terminates if a detection rule R_{ij} determines that

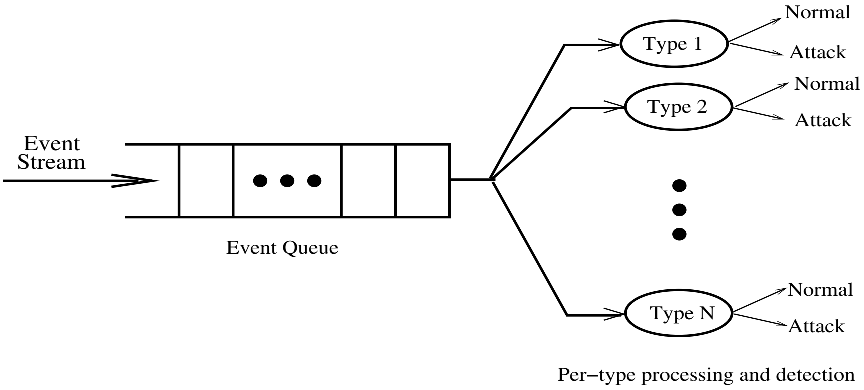


Fig. 1. The IDS Processing Flow. All events are directed to a common queue, but the nature of the service performed on each event depends on event type.

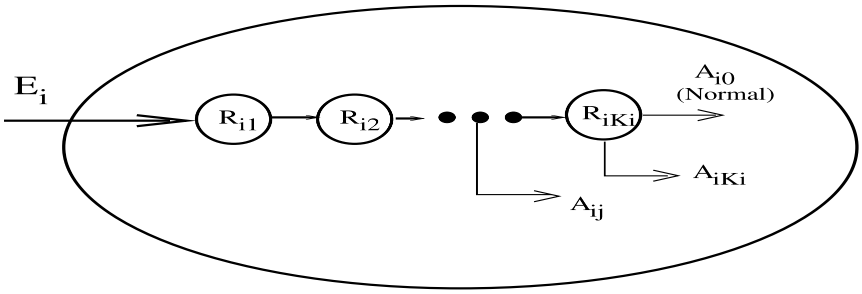


Fig. 2. Processing of events of type i. That tasks include preprocessing, rule-checking, and logging. They are applied serially.

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 - [13]) 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, $i = 1, 2, \cdots, N$ is given by:

$$T(\mathcal{P}) = (\sum_{i=1}^{N} m_i T_i) + T_{i'} \tag{2}$$

where T_i denotes the expected service time for a record of type i. The system time corresponds to the time interval elapsed between an audit record entering the system and a decision being made about the presence or absence of an attack in the event. We call it the response time of the IDS. We are interested in the "worst case" when the queue is close to full. In such a case, we have $\sum_{i=1}^{N} m_i T_i \gg T_{i'}$, that is, the queuing time dominates the service time for a typical event. Equation (2) becomes:

$$T(\mathcal{P}) = \sum_{i=1}^{N} m_i T_i, \quad \text{where} \quad T_i = \sum_{j=0}^{N_i} p_{ij} T_{ij}, \quad \text{and} \quad T_{ij} = \sum_{\ell=1}^{j} t'_{i\ell} \tag{3}$$

T_{ij} denotes the service time of an event of type i which is matched by detection rule R_{ij}. Here, $T_{i0} = T_{iK_i}$, since, as depicted in Figure 2, an event of type i which is labeled as normal has the same service time of an event which is matched by the last detection rule R_{iN_i}. Note that here we include ("factor in") the time of common preprocessing and logging tasks for event type i 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:

$$T(\mathcal{P}) = \sum_{i=1}^{N} \sum_{j=1}^{N_i} u_{ij} t'_{ij},$$

$$\text{where} \quad q_{ij} = 1 - \sum_{\ell=1}^{j-1} p_{i\ell}, \text{ for } j \geq 1, \quad q_{i1} = 1, \quad \text{and} \quad u_{ij} = m_i q_{ij} \tag{4}$$

2.3 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 reassembly 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 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). The IDS value $\mathcal{V}_{ij}$ (see Equation (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.

Our 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, we need to solve the following problem:

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

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

$$\text{where} \quad x_{ij} = 1 \;\text{ if } R_{ij} \text{ is active in } \mathcal{P} \text{ and } \; x_{ij} = 0 \;\text{ otherwise} \tag{5}$$

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., [18, 22]) 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, we omit the time and value in such a situation).

Instead of requiring exact measurements of the parameter values (e.g., p_{ij}) when solving Problem (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.

2.4 Static Configuration vs. Adaptation

The above optimization problem is often implicitly studied in IDS development and (initial) deployment time. That is, 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 2.4 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.

We note that a solution to Problem (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 [25, 23], 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 $\mathcal{V}_{kl} < \mathcal{V}_{ij}$.

We define *performance adaptation* 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. We consider the problem of improving the capability to deal with high-speed and high-volume audit data stream (e.g. making IDS more efficient thus reducing $(T(\mathcal{P}))$ an orthogonal issue to performance adaptation. In [8], we describe our work in designing and implementing IDS on programmable network processors. We believe that by carrying out intrusion detection tasks right at the network processor level, an IDS can keep up with very high speed and high volume traffic.

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 (5). In Section 3, we will detail our performance monitoring and adaptation techniques and their implementation in prototype real-time IDSs.

Experiments with IDSs We conducted experiments to study how IDSs perform under stress and how a statically configured IDS can be overloaded by a determined adversary. We used Snort [27] Version 1.8.6 with a subset of its latest ruleset and a number of important preprocessors (e.g., `frag2`, `stream4`, `http-decode`, `unidecode`, `telnet`, `telnet decode`, `rpc decode`, and `portscan`)[4],

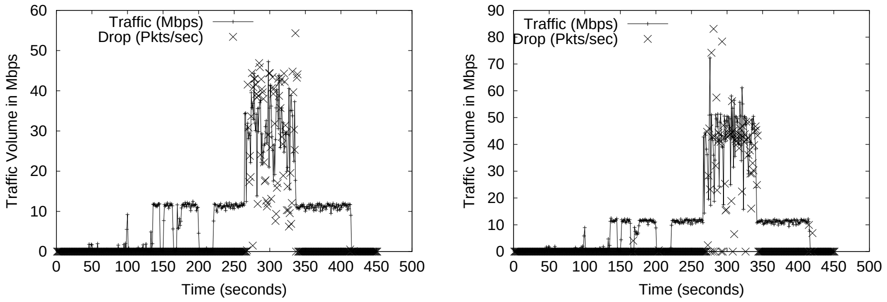
[4] We used the recommended options "`snort -c snort.conf -b -A fast`'' to make it run as fast as possible.

as well as Bro [23] Version 0.7a90. Snort is very lightweight: it performs packet-based analysis (its `TCP` stream re-assembly is incomplete). Bro, on the other hand, performs full packet re-assembly, connection stateful analysis, and even keystroke editing. In Bro, 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. 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.

We ran Bro with the standard scripts that come with the distribution, plus a few additional rules to detect our test attacks. We also set the configuration file to load the `http`, `udp`, `icmp` 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 our experiments, we used "(`tcp`[13]&0x7!=0) or (port `ftp`) or (port `telnet`) or (dst port 80 or dst port 8080) or (`udp` port 53) or (`icmp`)", 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.

Our experiments were conducted using LARIAT [28], 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. It provides many ways to configure background traffic and attack generation, and facilitates repeatable controlled experiments. We obtained the LARIAT software from MIT Lincoln Lab, and have built a network testbed based on LARIAT. The testbed, with several traffic generators, can produce (simulate) intranet and (both in-bound out-bound) Internet traffic. We use this testbed in our research to simulate (thousands of) virtual hosts and virtual sessions, and create high-speed and high-volume network traffic to test our algorithms and prototype IDS.

The background traffic in our experiments was generated using LARIAT scripts based on traffic profiles. The overload attack experiments performed on the IDSs were done in two stages: in the first stage, we overload the IDS by sending a huge amount of traffic, and in the second stage we launch a critical attack. 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. This kind of scenario works for



(a) Original Snort

(b) Original Bro

Fig. 3. Performance of IDSs under stress: when traffic volume reaches a certain point, IDSs drop packets indiscriminately.

the attacker especially when the attack involves very few packets, for example a buffer overflow exploit or the `WEB-IIS CMD.EXE` attack[5]. 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. In our experiments, the traffic surges (spikes) is a combination of `udp` packets (to `DNS` port) and `udp` packets to a user port. 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`[6] version 1.1; thereby ensuring exactly the same traffic conditions for all experiments.

As shown in Figure 3, the results from our experiments showed that both Snort and Bro can be overloaded when the traffic volume was increased to a certain point (> 40 Mbps) and will 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). The exact IDS configurations, traffic conditions, and attack strategies leading to these results are not very important. The point here is that an IDS can be overloaded to drop packets. Unless it can detect such conditions and quickly reconfigure appropriately, it can miss the critical attacks (that can otherwise be detected) with high probability. Thus, instead of being statically configured, a real-time IDS needs to provide performance adaptation via dynamic reconfiguration.

---

[5] 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.

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

2.5 Practical Considerations

We start by describing the guidelines for obtaining estimates for the various parameters in our method. 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 3.3) 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 2.3, 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") [9, 21, 3]. One needs to first define a site-specific *attack taxonomy*. For example, attacks can be first categorized by results (e.g., root access, user access, denial-of-service (DoS), and probing), then further by techniques (e.g., DoS by resource consumption or crashing), and still further by targets (e.g., a server or a desktop) [15]. Then knowledge about the intrusions (e.g., buffer-overflow is usually more damaging than DoS) and assets (e.g., the same intrusion to a server is more damaging than to a desktop) can be used to *qualitatively* measure the damage costs in relative scales [21, 15]. 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 [15]. 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 later in the paper 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.

One may argue that dealing with prior probabilities is always too hard (despite our best effort to simplify the matter), and instead of attempting to solve the optimization problem, an IDS should just use some simple reconfiguring techniques. We note that such "simple" techniques can be derived from the optimization problem by simplifying certain parameters and computations. For example, when stressed or overloaded, an IDS can cease to accept certain types of audit data (e.g., `icmp` and `udp`), and only provide full analysis for a small number of types of events (e.g., analyze `telnet`, `ftp`, and `http` only). In this example, the IDS regards the analysis tasks for certain events as more important than those for the others. Implicitly, the IDS has a ranking order of the values of the analysis tasks. This IDS reconfiguration is equivalent to solving Problem (5) by using only the values (their rankings) and setting the time required for the less important (to be excluded) events to an artificially large value (e.g., D_{max}).

If these simple techniques seem to be adequate in practice[7], what then is the value of our elaborate analysis here? We believe that it is important to provide a formal understanding of IDS performance analysis and adaptation, so that one can follow the optimization principles discussed here to improve upon various simple techniques to provide better IDS value. We also believe that measurement techniques and tools will become available and matured, and the full potential of optimization techniques will be realized.

3 Enabling Performance Adaptation in Real-Time IDS

In this section, we discuss how to enable real-time IDS to provide performance adaptation. We describe prototype systems, and present experiment results.

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 (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 [34]. Our research is com-

[7] We have no evidence that this is indeed the case.

plimentary 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 3.3), and alarm correlation and reduction, which are considered very important and desired IDS features [30]. In our approach, the back-end can also carry out some analysis tasks shed from the front-end.

3.1 Performance Monitoring

According to Problem (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 2.3, 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 [23], or an infinite-loop due to implementation errors), and can take immediate action such as activating another (replacement) front-end module.

3.2 Dynamic Reconfiguration

As described in Section 2.1, 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 (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 2.5, 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 3.3) 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.

3.3 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), timestamp, 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 3.2). The target information is very useful to determine (if necessary) what portion of the audit data the front-end can stop capturing. We are actively studying how to automatically update scenario information and discover new scenarios.

3.4 Prototype Systems

We next describe prototype systems, and present some experiment results. 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 perform in 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.

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 2.1. 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 2.3. $T(\mathcal{P})$ is equivalent 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 (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. We implemented a dynamic programming Knapsack algorithm. 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. 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 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 terminate an offending connection. 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. Presently, we only have very primitive scenario analysis functions.

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., de-fragmentation) 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`, `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.

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. 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.

Experiments We conducted a set of experiments, similar to those described in Section 2.4, to study the performance of our prototype Adaptive Bro and Adaptive Snort. We replayed the same traffic as explained earlier using `tcpreplay`. 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 [21, 15]. 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.

Figure 4 shows the behavior of the Adaptive Bro and Adaptive Snort when overloaded with the same background and flooding traffic used before. We describe some details as follows. 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

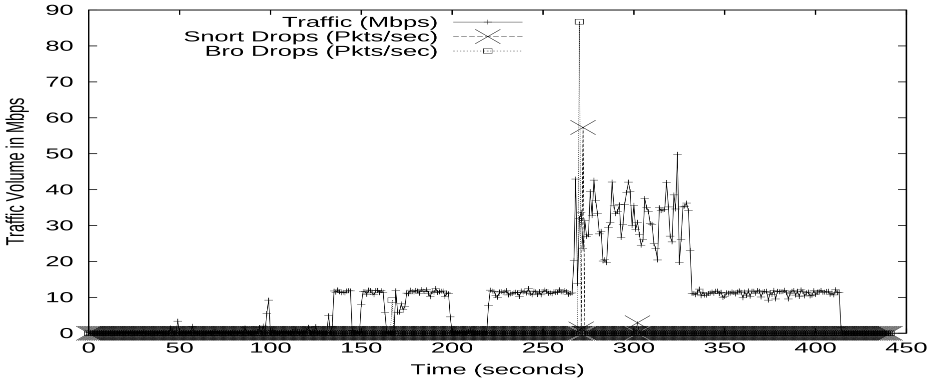


Fig. 4. Behavior of adaptive IDSs under stress: when under stress, it changes to a new configuration to minimize delay and data loss.

80 or dst port 8080) or (port `imap`) or (`udp` port 53) or (`icmp`)" as before. For Adaptive Snort, the initial configuration consisted of a subset of the default rule set (277 rules) coming with the distribution and with the important preprocessors (i.e. `frag2`, `stream4`, `portscan`, `http-decode`, `unidecode`, `rpc-decode`, `telnet-decode`) activated. 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, but the quick reconfiguration avoided further packet drops. This happens at time $t = 270$. This is different from the Original Bro, as shown in Figure 3, 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 (5)). The reconfiguration ended up dropping all tasks for `udp` and hence the filter for picking up `udp` traffic. 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.

We also experimented with the same overload attacks described in Section 2.4 against Adaptive Bro and Adaptive Snort. The systems detected the WEB-IIS CMD.EXE exploits more than 90% of time (19 out of 20) for Bro and 100% of the time for Snort (20 out of 20).

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.

4 Related Work

The problems of real-time network-based IDSs being evaded (e.g. with "ambiguous" packets), overloaded, and crashed were first discussed in [25, 23]. Evasion can be prevented if an IDS uses stateful analysis and a network traffic *normalizer* [12]. Although high-end hardware platforms can be used for data capturing to ensure "no packet filter drops" [23], an IDS, with the often site-specific ID logics implemented as application-level software, can still be overloaded due to high volume filtered events. Paxson suggested that load-shedding may help a real-time IDS defend against overload attacks [23]. Several enterprise-wide and Internet-wide distributed IDSs [24, 35] and agent-based architectures [4, 11] have been proposed to address the issues of detection coverage and workload distribution. For example, in EMERALD [24], ID modules are deployed and configured in a hierarchical fashion according to the enterprise network topology. More recently, Kruegel et al. proposed a partitioning approach for intrusion detection in high speed network environments [14]. 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. 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. Therefore some adaptive approach has to be incorporated. Our research complements 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 [10, 15]. We use cost-benefit analysis to determine the best IDS configurations given the resource constraints.

Operating systems also need load-shedding techniques (e.g., [7, 20]) to support real-time and multimedia applications. Performance monitoring and Quality-of-Service adaptation approaches have been studied in other domains, e.g., network servers (e.g., [1]), and middleware (e.g., [16]). We can learn from these studies and design techniques for monitoring IDS performance and shedding workload to satisfy the real-time requirements of high priority detection tasks. Although load-balancing is typically used in a different context than load-shedding, we can still learn from the research in this classical problem. For example, in addition to measuring resource utilization, it is necessary to test the application response in order to determine the "meaningful" system load. In our approach, the manager tests the IDS response time using a simulated attack. We can also learn from the vast amount of research in dynamic real-time system scheduling (e.g., [32, 17]), and develop techniques to optimize real-time IDSs.

Attack tree analysis [29] is an off-line process. We are interested in recognizing run-time scenarios and predicting likely forthcoming attacks. Similar to the activity graphs in GrIDS [31, 6], we use a scenario graph and network topology to detect scenarios.

5 Conclusion

Providing performance guarantee (assurance) should be the key requirement of IDSs, and security products in general. In this paper, we provided 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. Results from experiments thus far validate our motivation and approach.

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.

Acknowledgments

This research is supported in part by grants from DARPA (F30602-00-1-0603). We wish to thank Vern Paxson of ICSI/LBNL for help on Bro, and Lee Rossey and Richard Lippmann of MIT Lincoln Lab for help on LARIAT.

References

1. T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 2001. to appear.
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. S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, and D. Zerkle. The design of GrIDS: A graph-based intrusion detection system. Technical Report CSE-99-2, U.C. Davis Computer Science Department, Davis, CA, 1999.
7. C. L. Compton and D. L. Tennenhouse. Collaborative load shedding for media-based applications. In *International Conference on Multimedia Computing and Systems*, Boston, MA, May 1994.
8. D. Contis, W. Lee, D. E. Schimmel, W. Shi, A. Thomas, Y. Zhang, Jun Li, and C. Clark. A prototype programmable network processor based ids. submitted for publication, March 2002.
9. D. Denning. *Information Warfare and Security.* Addison Wesley, 1999.
10. 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.
11. 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.
12. 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.
13. L. Kleinrock. *Queuing Systems, Vol. 1: Theory.* John Wiley & Sons, Inc., 1975.
14. 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.
15. 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.
16. L. Liu, C. Pu, K. Schwan, and J. Walpole. Infofilter: Supporting quality of service for fresh information delivery. *New Generation Computing Journal*, 18(4), August 2000.
17. C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2000.
18. S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons Ltd., 1990.
19. S. McCanne, C. Leres, and V. Jacobson. `libpcap`. available via anonymous ftp to ftp.ee.lbl.gov, 1994.
20. J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the Sixteen ACM Symposium on Operating Systems Principles*, October 1997.
21. S. Northcutt. *Intrusion Detection: An Analyst's Handbook.* New Riders, 1999.
22. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity.* Prentice-Hall, Inc., 1982.

23. V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), December 1999.
24. 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.
25. 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.
26. 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.
27. 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.
28. 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.
29. B. Schneier. *Secrets & Lies: Digital Security in a Networked World*. John Wiley & Sons, Inc., 2000.
30. G. Shipley and P. Mueller. Dragon claws its way to the top. In *Network Computing*. TechWeb, August 2001.
31. S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS-a graph based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*, 1996.
32. J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, June 1999.
33. SunSoft. *SunSHIELD Basic Security Module Guide*. SunSoft, Mountain View, CA, 1995.
34. 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.
35. 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.